LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 746-751
cop. 2

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

JUL 1 3 1978

JUN 1 2 REC'D

L161—O-1096

Digitized by the Internet Archive
in 2013

http://archive.org/details/interactivecompi748tind

510.84
Il6r
no. 748 UIUCDCS-R-75-748

Math

THE LIBRARY OF THE

SEP 8 1975

UNIVERSITY OF ILLINOIS

AN INTERACTIVE COMPILE-TIME DIAGNOSTIC SYSTEM

BY

Michael H. Tindall

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-75-748

AN INTERACTIVE COMPILE-TIME DIAGNOSTIC SYSTEM*

BY

MICHAEL H. TINDALL

October 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by an International Business Machine Corporation educational fellowship and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, October 1975.

ABSTRACT

Let the following be a few of the characteristics of an interactive compiler system: the user's program is interactively entered via a keyboard and display-screen timesharing terminal (e.g., the PLATO IV CAI system); the program is scanned and syntactically parsed as it is input by the user; programming syntax errors are signalled as soon as detected by the parser, and the user must correct the errors before proceeding (this implies that no right context of the user program is available for examination by the error analysis system).

Under these assumptions, the following interactive diagnostic system can be devised. The system is "automatic," that is, it is driven by the compiler's parser tables. The error system behaves like a consultant by suggesting "possible corrections" of the program to the user, and at any time the user can proceed to fix the program or request further suggestions. In addition, the "possible correction" diagnostic suggestions can refer to not only individual "tokens" in the user's program, but also higher-level constructs, such as "expressions," "array bounds," etc.

Key Phrases: Compilers, error analysis, interactive compiling, parsers, transition diagrams.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to his thesis advisor, Professor Thomas Wilcox, for his many suggestions and critical guidance throughout the work on this thesis.

Thanks are also in order for Professor Alan Davis and J. Mike Milner for their assistance with the implementation of the prototype diagnostic system on PLATO IV. A sincere thank you is also given to the other members of the thesis committee: Professors J. Nievergelt, H. G. Friedman, M. D. Mickunas, and E. M. Reingold for their support. Professor Mickunas in particular, as well as Professor W. Hansen are thanked for their helpful comments about this thesis.

Finally, my wife, Elaine, must be thanked; it was only through her patience and support that this work was completed.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION.	1
	1.1 The Problem	1
	1.2 The Compiling Environment.	2
	1.3 Compile-time Error System Constraints and Goals	5
	1.4 Organization of the Thesis	8
2	SURVEY OF ERROR CORRECTION AND RECOVERY TECHNIQUES	10
3	THE TRANSITION DIAGRAM PARSING MODEL.	20
	3.1 Introduction	20
	3.2 General Characteristics of the Transition Diagram Model	20
	3.3 Formal Description of the Transition Diagram Model	29
	3.4 Description of the Implemented Parser.	30
	3.5 Compiler Environment when Parser Signals Error.	32
4	A MODEL FOR DIAGNOSTIC INTERACTION WITH THE PROGRAMMER	34
	4.1 Goals for Diagnostic Interaction	34
	4.2 Suggesting "Possible Corrections" for the Programmer	35
	4.3 Algorithmic Notation	42
	4.4 Algorithm for Diagnostic Interaction	42
5	DETERMINING "POSSIBLE CORRECTION" SUGGESTIONS.	44
	5.1 Introduction	44
	5.2 A First Model for Determining "Possible Correction" Suggestions	44
	5.3 Extended Suggestion Model.	63
	5.4 Overall Control of the Extended Model.	88
6	CONCLUSIONS AND FUTURE RESEARCH	93
	6.1 Summary.	93

TABLE OF CONTENTS (Continued)

Chapter Page

6.2 Evaluation. 95
6.3 Future Research 101
6.4 Conclusion. 102

LIST OF REFERENCES 103

APPENDIX

I GENERAL OPTION TREE EXAMPLES 106

II EXAMPLES OF "POSSIBLE CORRECTION" SUGGESTIONS FOR PL/I 125

VITA. 166

LIST OF TABLES

Table		Page
5.1	Token Options for States of Transition Diagram E1	59
5.2	Error Interpretations for Transition Diagram E1	60
5.3	Token Options for States of Transition Diagrams E2 and E3.	61
5.4	Error Interpretations for Transition Diagrams E2 and E3.	62
5.5	General Options for States of Transition Diagrams E2 and E3.	81
5.6	Extended Error Interpretations for Transition Diagrams E2 and E3	82
5.7	General Options for Diagrams E6 - E9.	86
5.8	Some Extended Error Interpretations for Diagrams E6 - E9	87

LIST OF FIGURES

Figure Page

3.1 - Simple transition diagram example. 23

3.2 - PL/I "GOTO" statement transition diagram . . . 23

3.3 - Invoking transition diagram example. 25

3.4 - PL/I "IF" statement transition diagram 26

3.5 - Multiple return transition diagram example. . . 27

3.6 - PL/I conditional expression example 28

3.7 - PL/I <conditional-expression> transition diagram 29

4.1(a)-Example of suggestion messages. 36

(b)-Example 37

(c)-Example 37

(d)-Example 38

4.2(a)-"Detail" suggestion messages example. 39

(b)-"Detail" example 40

(c)-"Detail" example 40

(d)-"Detail" example 41

(e)-"Detail" example 41

4.3 - Algorithm 'USER_INTERACTION' 43

5.1 - The "transit" function 52

5.2 - "Token" MODIFY routine 57

5.3 - REPARSE routine. 58

5.4 - MODIFY routine, version 2 69

5.5 - Final USER_INTERACTION routine. 75

LIST OF FIGURES (Continued)

Figure Page

5.6 - Final REPARSE routine. 76

5.7 - Final MODIFY routine 78

5.8 - Final MOD_TREE routine 80

5.9 - Transition diagrams E6 - E9. 84

5.10 - ERROR_SYSTEM_CONTROL routine 92

Chapter 1

INTRODUCTION

1.1 The Problem

Novice computer programmers, such as students taking an elementary computer programming course, spend a substantial fraction of the total time required to develop a computer program removing compiler-detected syntactic errors. The compilers used by these elementary programmers must provide informative and understandable diagnostic messages when compile-time errors are discovered.

At the present time, there are a few compilers available that operate in a batch-mode environment and provide reasonably good diagnostics (for example, PL/C (Conway and Wilcox [Con., '73]) and WATFIV (Cress, et al. [Cre., '70]). However, for a compiler that operates in an interactive timesharing environment, that is, an incremental compiler, the diagnostic assistance that can be provided to an elementary programmer can be much more informative than is possible in a batch-oriented compiler. It is possible for an interactive diagnostic system to provide various "levels" of error messages about a particular error situation; some messages may suggest fairly general ways of fixing the program, while others provide very explicit detail about possible corrections of the

error. Using the fact that the programmer is preparing the program via an interactive computer terminal, the diagnostic system can present error messages to the programmer and, based on the programmer's response, tailor subsequent, more detailed or refined error messages to the comprehension level of the particular programmer.

This thesis describes a syntactic error diagnostic system that is intended for use in a highly interactive, syntax table-driven compiler system. The primary goal of this diagnostic system is to provide good diagnostic messages in a form and terminology that an elementary programmer can understand. It is assumed that the novice programmer is compiling the program on an interactive timesharing computer and compiler; therefore, the programmer can, to some extent, control the diagnostic messages that are prepared and displayed. In addition, the error analysis algorithm makes complete use of the syntax tables, and thus any error analysis is performed "automatically" by the compiler system with very little additional work required of the language designer beyond specifying the syntax table itself.

1.2 The Compiling Environment

Recently, a project at the University of Illinois at Urbana-Champaign has been under way to automate the teaching of the basic computer science courses by using the PLATO IV computer-based

education system [Nie., '74]. The curriculum that has been implemented teaches elementary computer science programming language concepts and constructs to classes of students with highly-varied backgrounds. Specific detail on a variety of languages (e.g., PL/I, FORTRAN IV, COBOL, BASIC) is available; students progress at their own speeds through a fairly flexible course structure [Ela., '75]. An important part of the system is an on-line interactive compiler in which a student can easily and conveniently try out new programming language constructs immediately after learning about them in an instructional lesson [Wil., '73].

A number of design criteria for a compiler system emerge from examining this interactive environment. First, the compiler should be as interactive as possible to utilize the PLATO IV system effectively and to maintain a desirable computer-aided instructional environment for the student. To accomplish this, the compiler compiles character-by-character; that is, each single key pressed by the student using the compiler is examined immediately as the student types it in. Thus, the compiler keeps up completely with the student's preparation of a program.

Another result of this immediate, incremental compiling technique is that syntax errors are detected and signalled as soon as possible by the system. The student must correct the error before being allowed to enter the remainder of the program.

The student is able to edit the program by moving a

cursor through the program on the PLATO graphics screen. The compiler moves with the cursor, compiling when the cursor moves forward in the program, and backing-up ("uncompiling," i.e., resetting the lexical and syntactical analyzers to previous states) when the cursor moves backward in the program. To facilitate the "uncompiling" capability, a parser-action history is created and maintained as the program is compiled; this history records any changes that are made in the parser's environment for each token as it is parsed. Then, to "uncompile" a token, it is necessary to undo only changes recorded in the history for that token. This parser history is also very useful for the error diagnostic system, since it contains a complete description of what actions the parser system has taken.

A second design criterion for the compiler is that it be multilingual. To accomplish this, the compiler is completely table-driven; to allow another language to be recognized by the compiler system and used by students, a language designer must merely fill in a new set of tables and provide an execution supervisor system for the actual interpretive execution of compiled programs.

A third design criterion for the compiler is that it provide a high and sophisticated level of error diagnostics for the student. Since the intended users of the compiler are beginning student programmers, the error messages must be direct and to-the-point. A system has been designed and implemented to handle errors

that are detected during the interpretive execution of a student's program [Dav., '75]. The present thesis is concerned with diagnosing errors that are detected during the compiling phase of program preparation.

1.3 Compile-time Error System Constraints and Goals

One very severe constraint for the diagnostic system is that any messages and algorithmic analysis must reference <u>only</u> those tokens in the program up to and including the token at which an error was detected. In an interactive timesharing environment, unlike any batch-oriented compiler environment, right context for an error is not available for examination most of the time. This is because the interactive compiler detects and flags syntactic errors as soon as the programmer types in an offending token. The only time when any right context for an error is available is the situation where the programmer enters a part of the program into the system and then moves the cursor backward in the program ("uncompiling" the tokens that are backed over); if the programmer then attempts to make a syntactically illegal modification somewhere in the middle of the program, all of the backed-over tokens are available as right context for the error. However, since the programmer was in the middle of making some change to the program when the error was detected, it is unlikely that the right context has anything to do with the newly-formed left context of the error;

the input string will usually be in a transitory, incomplete form. Therefore, the general, more restrictive case that must be considered is where no right context is available for analysis.

Another constraint is that the error system must operate "automatically" and be as language independent as possible. To achieve this independence, the error analysis system must be able to determine any messages by examining only the input token string, any compiler symbol tables, and the parser table. To keep the compiler system multilingual, no language-specific, *ad hoc* error analysis techniques can be used.

There are a number of goals for this compile-time error diagnostic system. One is that all diagnostic messages be understandable to elementary student programmers. In particular, all messages must be stated in the terminology that is associated with the particular programming language being taught to and used by the student. This means that if things like "statements," "arithmetic expressions," and "array bounds" are important concepts in a programming language, then diagnostic messages should refer to these higher-level constructs when appropriate; however, the actual specific detail pertaining to the higher-level constructs should always be available to the programmer if it is needed or requested.

Another goal for the diagnostic system is that it must be highly interactive and responsive to the student. As previously mentioned, a high level of interaction will enable the diagnostic

system to be more effective for the student than would be possible with a noninteractive system. In addition, all of the instructional lessons that a student takes on the PLATO IV system are highly interactive and require student input frequently to be completed successfully. The overall compiler system that has been described is very interactive and is constantly monitoring the progress of the student as a program is being prepared. Therefore, the diagnostic system should also be interactive in order to conform with the rest of the student's environment.

A further goal for the diagnostic system that it must be "feasible" for implementation within the PLATO IV system. The PLATO IV system (Alpert and Bitzer [Alp., '70]) was designed as a large-scale computer-aided instruction system to support as many as 1000 simultaneous users. The student terminal contains a plasma display panel as its main output component and a keyboard as its input component. The panel permits quick display of both textual information and arbitrarily complex graphic output. (This capability can be used very effectively in conveying useful diagnostic information about an error to a student.) However, because of the large number of users allowed by the PLATO IV system, the amount of central processor computing time allowed per user is very restricted. Therefore the error analysis algorithms used to determine diagnostic messages for the student must be fairly efficient and not require too much central processing time.

The overwhelming goal and emphasis for this diagnostic system is that it behave like a "consultant," that is, be able to diagnose programming errors and help the student understand the errors. As much of this "consultant" as is possible should be built into the compiler system and be provided automatically for different programming languages. It is the activities and operations of this "consultant" with respect to syntactic and context-sensitive semantic errors that is the topic of this thesis.

1.4 Organization of the Thesis

Chapter 1 has presented an overview of the problem of providing good syntactic error diagnostics in an interactive compiling environment. The characteristics of a particular student-oriented interactive compiler system were discussed and a few of the goals and constraints imposed on a diagnostic system by such a compiler system were mentioned.

Chapter 2 will present a survey of some of the previous work that has been done on error correction and recovery techniques. Chapter 3 gives a description of the transition diagram parser model that is used by the compiler system and on which the proposed diagnostic consultant system is based.

Chapter 4 presents a model of an interactive diagnostic system that is appropriate for use in an interactive compiler. Chapter 5 then describes the algorithms and operations of the

proposed error analysis system, and some final concluding remarks are given in Chapter 6.

Chapter 2

SURVEY OF ERROR CORRECTION AND RECOVERY TECHNIQUES

This chapter will present a brief survey of a number of papers in the area of syntactic error detection, correction and recovery. Other good surveys appear in the papers by Rhodes [Rho., '73, pp. 9-25] and LaFrance [LaF., '71, pp. 2-7], and LaFrance's paper contains an unusually complete bibliography of the relevant literature prior to 1971.

It is important to note that each of the following systems was intended for batch-oriented analysis, as opposed to an interactive timesharing analysis. As a result, the goal of each system is to properly diagnose the syntactic error, or at least make a guess as to the cause of the error, and then attempt to somehow "correct" the user's program by modifying the input string or perhaps adjusting the parser's stack appropriately, and thus allow the compiler to "recover" from the error and continue parsing the remainder of the program.

As pointed out by Rhodes [Rho., '73, pp. 10-11], one of the more common error analysis techniques that is used, particularly in earlier systems, implements the so-called "panic mode" of error recovery. With varying degrees of sophistication, the "panic mode" recovery operates by popping the stack and advancing the input until a "safe" environment is achieved. This method is simple

to use, but the errors contained in that portion of the text which is skipped are not detected. The emphasis of the "panic mode" method is simply to provide a technique which allows the parser to recover from the error and continue parsing more of the input string.

One of the earliest papers on an automatic error system came from E. T. Irons [Iro., '63]. Irons' parsing algorithm constructed all possible legal parses for the input string in parallel. An error was signalled when none of the parses could be extended one more symbol. In this event, Irons' error system built a list of all the possible successor symbols from each parse directly before the error point. Then the input was discarded until a symbol was found that appeared on the list. Finally, a string of symbols was created that, when inserted at the error point, would allow the parse to continue. This achieved the effect of local insertion, deletion or replacement of symbols to repair the error.

A more sophisticated version of the "panic mode" method was proposed by Leinius [Lei., '70]. This work was primarily based on a simple precedence parser [Wir., '66]. The recovery system attempts to find the smallest substring containing the error which can be reduced, and then forces the appropriate stack reduction and input scanning to occur.

Leinius also described the application of his technique to other parsing systems, and a master's thesis by L. R. James

[JaL., '72] discusses a detailed system using a bottom-up LALR(k) (lookahead LR(k)) parser [DeR., '71]. As the parse progresses, "configuration sets" are produced; these configuration sets consist of the right-hand sides (RHS) of productions in the grammar (see [Aho, '72] for a description of grammars, productions, terminal and nonterminal symbols), with the following transformation: each RHS contains a distinguished symbol not appearing in the language (e.g., ".") which indicates how far in the RHS the parse has progressed as of this configuration set. As input terminals are read, the configuration set is updated by moving the "." one symbol to the right in the elements which apply. When the "." is moved so that it is directly to the left of a nonterminal in a production, then this configuration set is pushed onto a state stack and the configuration set for that new nonterminal is used. When the "." reaches the far right of a RHS in the configuration set, reductions on the stack can take place and the elements in the configuration set which used the reduced symbol are updated.

An error occurs when no RHS in the configuration set applies to the input symbol. The error algorithm attempts to recover as follows: the parser state stack is popped and the input string is skipped until a configuration set is found which is associated with a nonterminal that can immediately preceed the next terminal symbol in the input. Finally the configuration set is updated to indicate that the nonterminal has now been parsed, and

the normal parse continues. This algorithm will always terminate since the entire program can be considered as one large string which reduces to the start symbol of the grammar.

In the last few years, a number of error systems have been proposed that emphasize the "correction" of the error that is made, as well as the recovery of the parser. The systems generally attempt to examine the input tokens that are located near to where the error was detected and use this "context" as the basis of the correction that is made to the program. An interesting technique based on a simple precedence parser is proposed by Graham and Rhodes ([Gra., '73] and [Rho., '73]). As input symbols are read, they are either immediately stacked, or they cause the stack to be reduced and then they are stacked, depending upon the precedence relation between the input symbol and the symbol on top of the stack. Two types of errors can be detected in this system: either no precedence relation holds between the input symbol and the symbol on top of the stack (called a "character pair" error), or else a potential RHS is found, but no matching RHS exists in the grammar (called a "reduction" error).

When an error is detected, the error system begins phase one (the "condensation" phase) of two phases (the other is the "correction" phase). The "condensation" phase marks the location of the error in the stack, for later reference, as $l_?$. Then a "backward" move is initiated by assuming that a $>$ holds between the

symbol to the left of $1_?$ and the symbol to the right of $1_?$; this causes the stack to reduce if possible. Then the assumption is made that a < relation exists between the two symbols, and a "forward" move begins by passing control back to the parser. The "forward" move will end when another error is detected. (This may be a new error, or the reappearance of the old error in the form of a reduction error.) This new stack location is designated as $2_?$, and a "backward" move stack reduction is again performed. This ends the "condensation" phase.

The "correction" phase then attempts to change the parsing stack to correct the error situation. To do this, the "correction" phase will either replace or delete one of three sets of symbols on the stack: the set from the first < relation to the left of $1_?$ up to $1_?$, the set from that same < relation up to $2_?$, or the set from $1_?$ up to $2_?$. If it replaces one of these sets, it will replace it with a complete RHS from the grammar. The actual choice of what action to take is determined by a probabilistic pattern match, along with two "cost" vectors, which give the "cost" of inserting or deleting a given symbol from the input string. The total cost of deleting/replacing each symbol set is calculated and the cheapest is selected. (Note that the deletion/replacement must in turn produce a correct RHS on top of the stack.) The actual choice algorithm can be experimented with to yield the best results, as well as the entries in the "cost" vectors.

A system implemented by Lyon [Lyo., '74] operates in the following way. As the parser progresses, it constructs all possible parses: this includes all legal parses as well as all possible error parses. For example, not only is a possibly legal parse maintained, but also for each input token that is read, the parser carries along a parse that assumes that the token is an extra inserted symbol, as well as many parses that assume that some other symbol was deleted from in front of that token. In each parse, a count is kept of the number of errors that have occurred for that parse. When the input is exhausted, the parse which has the smallest error count is selected as the final parse; this algorithm is known as a least-errors recognizer. The results from Lyon's analysis indicate that his algorithm uses excessive amounts of execution time and storage space, thus relegating it, in its present form, to a theoretical rather than practical category.

Lyon's implementation is actually based on a system proposed by J. P. Levy [Lev., '71]. However, Levy's proposal is somewhat different from the system that Lyon developed. Levy describes a technique for moving back in the input a certain distance when an error is discovered, and then reparsing forward towards the error point as described in Lyon's algorithm; some heuristic routines then select one of the error reparsings to correct the program. It seems likely that this technique will eliminate the combinatorial problems encountered by Lyon's system.

An error system was proposed by J. E. LaFrance in 1970 ([LaF., '70] and [LaF., '71]). His system is based on a Floyd Production System parser [Flo., '63]. Briefly, this parser contains groups of productions. As the parse progresses, the parser moves from production group to production group; each group contains references to a set of productions that should be attempted at this point. If none of the productions apply, the error condition is raised. LaFrance's error system then generates a list of all possible two or three symbol "expected" strings from the production groups, beginning at the error point. Then each of these "possible" strings is compared (matched) to the next four symbols actually in the input string. The matching process consists of about twenty different comparisons such as "assume one symbol is missing" or "assume two symbols are interchanged." If a match is found, the source input string is repaired appropriately and the Floyd parser restarted. If no match is found, then the input string is discarded until a symbol is found which can follow a symbol in the stack; the stack is then reduced and the Floyd parser starts up at the place where it should be when the stack has just been reduced to that symbol. LaFrance's system is very good in a number of ways. It is a relatively efficient technique, since suitable encodings of the "possible" and the actual input strings allow the matching algorithm to be implemented efficiently. Also, the results of LaFrance's work indicate that corrections that are made to the

program are frequently the same corrections that the programmer would make in a similar situation, particularly for those error system corrections that were determined by a successful match having been found. However, if no match is found, LaFrance's system reverts directly back to the "panic mode" form of error recovery, with a corresponding degradation in the appropriateness of the corrections performed. Another criticism of LaFrance's system is that it examines only a few of the input symbols near the location of the error symbol, and ignores all the other context farther away from the error symbol. Examples can be given (for example, Rhodes [Rho., '73, pp. 25-26]) of error situations that occur in programming languages that cannot be properly corrected by LaFrance's technique.

One final system that should be mentioned is that of James and Partridge [JaE., '73]. They produced a system that uses many of the analysis techniques suggested by LaFrance, but with a strong adaptive slant: their system is designed to be able to modify itself somewhat to become more efficient or more effective. Their parser utilizes a binary tree structure, where each node in the tree contains some information plus a success and a fail link. The parser moves through the tree as the input is read, comparing the input symbol with the current node information and either taking the success or the fail link to another node. The system can restructure its own binary parse trees so that the most used path

occurs first in the tree. An error is detected if a fail is indicated, but the current node has a null fail link. This invokes the use of a separate "strategy" tree for error analysis. The strategy tree contains information nodes that suggest different matching strategies (very similar to those suggested by LaFrance). The system can also restructure the strategy tree so that more effective match strategies come first.

In summary, all of the techniques mentioned in this chapter were designed to operate in a batch-mode environment, where tokens both to the left and to the right of an error token are available for examination and analysis. Because of this, none of the systems can be directly applied in the interactive compiling environment described in section 1.3 of this thesis in which no right context is available for analysis. The "panic mode" systems are particularly ill-suited, since they operate primarily by skipping forward in the input until a token is found that can occur someplace after the actual error token. LaFrance's system (and also James and Partridge's system, since it is based on the LaFrance technique) is likewise poorly suited for this restrictive interactive environment since it analyzes the next few tokens following the error token. The works of Rhodes and Levy, however, are somewhat better suited to this environment, since both techniques begin by examining a substantial portion of the left context before the corresponding right context is analyzed. The work of Levy, which had

a large influence on the current thesis, is further described in section 5.2.1.

This concludes a look at some of the previous work in the field of error correction and recovery systems.

Chapter 3

THE TRANSITION DIAGRAM PARSING MODEL

3.1 Introduction

The syntactic analyzer used in this interactive compiler is based on the transition diagram systems first introduced by Conway [Con., '63], and later formalized by Lomet [Lom., '73]. The second section of this chapter will describe the most important characteristics and operations of a transition diagram parsing model; the third section will present a brief, more formal description of the parsing model defined by Lomet; the fourth section will discuss the actual parser table implementation that is used in the compiler system and which is referred to by the automatic error diagnostic system; the fifth and final section will summarize the parsing environment that exists when a syntactic error is detected. For more detailed information, the reader is referred to the above papers by Conway and Lomet, as well as a paper by Tindall [Tin., '75].

3.2 General Characteristics of the Transition Diagram Model

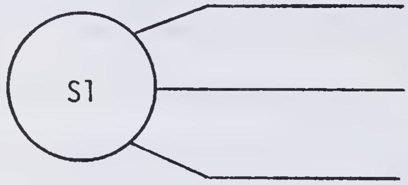
A transition diagram parser can be thought of as an expanded (stack-oriented) version of a simple finite state machine (fsm). A fsm maintains a "node" or "state" pointer, that is, at any time,

the fsm is said to be in some "state" waiting for the next input symbol to come in. When the next input symbol has been read, based on the attributes of the new symbol, the fsm then makes a "state transition" to a new state and waits for the next input symbol to be read. An entire input string is accepted if the input is exhausted and at the same time the fsm is in one of a set of special "final" states.

Similarly, the action of a transition diagram parser is to examine the "possible" or "acceptable" parsing options (determined by the current STATE and the transition diagrams) along with the current input token. Based on this information, the parser will accept the token by updating the STATE information and asking for a new input token, or reject the current input token and signal a syntactic error if the token does not satisfy any of the available parsing options.

Note that for the transition diagram model being described, all of the parsing options are defined in terms of "tokens" only, that is, the tokens that are accumulated and output from a lexical analyzer. This convention will remain the same for all of the work that is reported in this thesis unless otherwise stated. The parsing of input strings described in the rest of this chapter, as well as all of the error analysis manipulations discussed later in the thesis are all based on a "tokenized" input string.

The actions associated with this transition diagram model can be conveniently shown graphically as follows: Let



denote STATE "S1"; each branch out of a STATE corresponds to a possible syntax parse option for that STATE: these branches will always be labeled with their particular syntactic option requirements.

Figure 3.1 is an example of a simple transition diagram E1. This is interpreted as: if the parser is in STATE S1 and the current input token is "a," make the state transition to STATE S2. Otherwise, if the input token is "b," make the transition to STATE S7. Otherwise, an error has been detected, and the parsing error condition should be signalled. STATE S6 is a "final" state for diagram E1, as indicated by the null branch leading out of the S6 node: if the parser is in STATE S6 and the input is exhausted, then the entire input string is acceptable to the parser. The acceptable input strings for this transition diagram are "abcde," "ade," and "bfe."

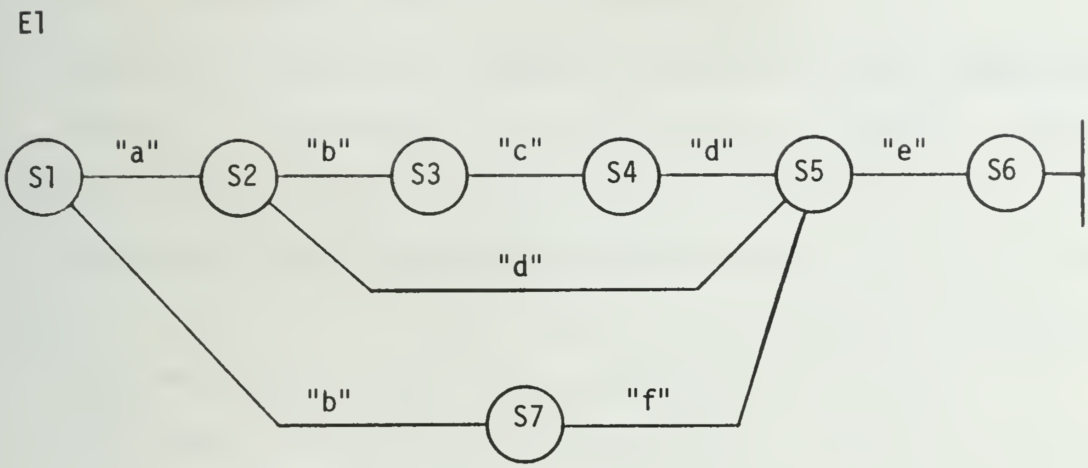


Figure 3.1 - Simple transition diagram example

A simple practical example is a form of a PL/I "GOTO" statement, which is shown in transition diagram form in Figure 3.2. Note that an input token can be referred to either literally or as some other combination of attributes such as [label-name], where [label-name] has an appropriate meaning in the parser environment.

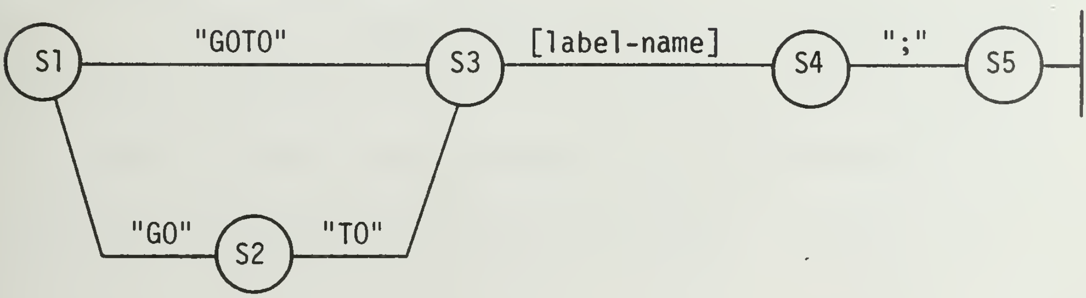


Figure 3.2 - PL/I "GOTO" statement transition diagram

So far in this discussion, all of the examples have been completely parsable within the framework of a simple finite state machine. However, the actual transition diagram model allows the label on a branch out of a node to refer to not just a single input token, but also a group of tokens that are specified by some other transition diagram within the parser system.

Figure 3.3 is an example of this more complex form of a transition diagram. In this case, if the parser is in STATE S2 in transition diagram E2, then the parser will save STATE S2 on a stack in the parser's environment and refer to diagram E3 to determine the attributes of the next acceptable input token. When the "final" state for diagram E3 is reached, the parser will then have completely accepted the input string specified by E3, and therefore, upon popping STATE S2 off of the parser's stack, will be able to legitimately make the state transition to STATE S3 and continue to parse as normal.

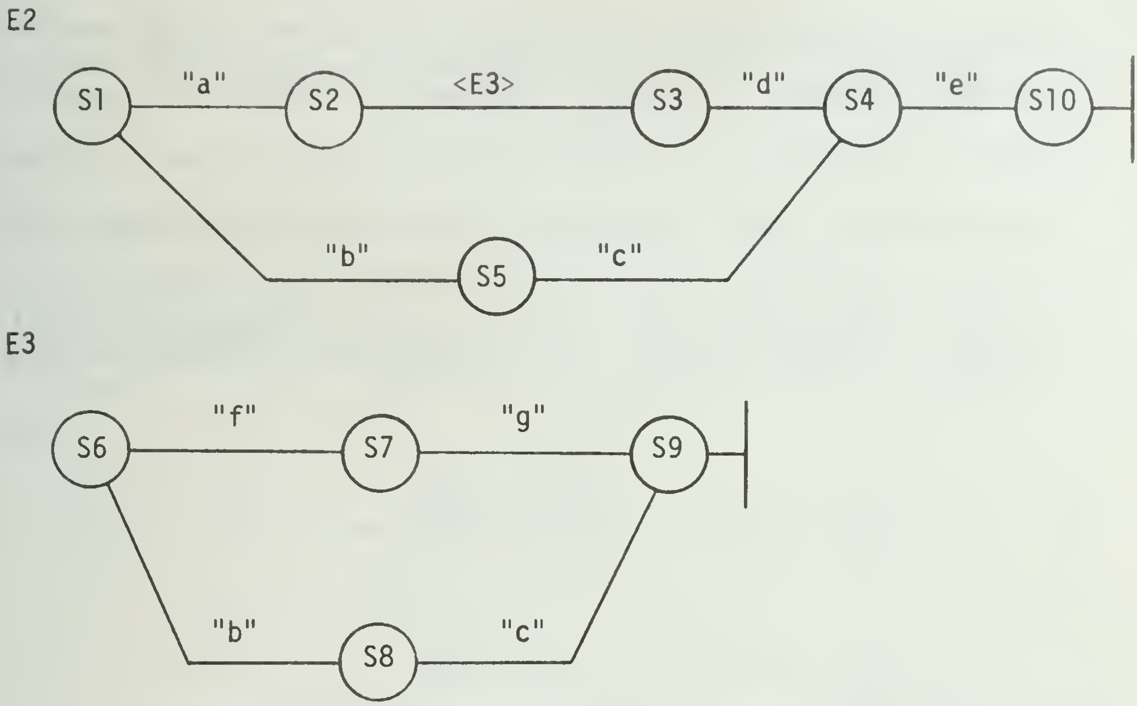


Figure 3.3 - Invoking transition diagram example

Although not explicitly illustrated by Figure 3.3, the "called" transition diagram (E3 in Figure 3.3) could refer to other transition diagrams in the system, or could in fact refer to itself. (However, E3 cannot be referred to from STATE S6, that is, "left recursion" must be avoided.)

A practical example that requires the invoking of other transition diagrams is the PL/I "IF" statement, shown in Figure 3.4. Note also that this transition diagram system can accept an entire "IF" statement either with or without the "ELSE" clause.

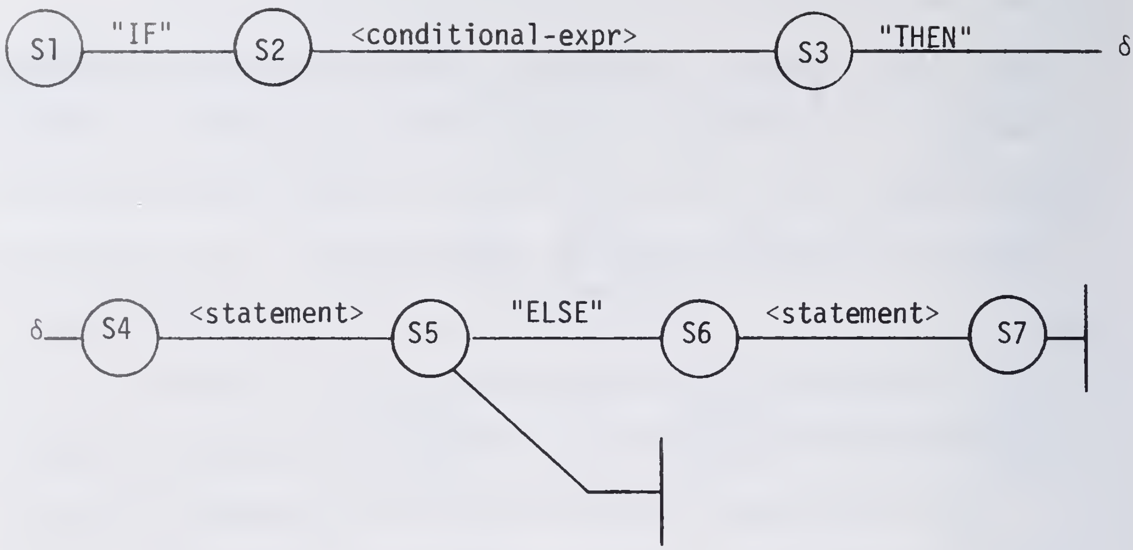


Figure 3.4 - PL/I "IF" statement transition diagram

One more example is needed to illustrate another important feature of transition diagrams: a transition diagram system which has been invoked by another transition diagram has the capability of returning to one of a number of possible STATEs in the invoking transition diagram. Figure 3.5 is an example of this situation. If the parser is in STATE S1, then diagram E5 is invoked normally using the parser's stack, and either input "a" or "c" is accepted, moving to STATE S7 or S8 respectively. Note that the parser is now in a "final" state for diagram E5; however, the small number given with the final state indicator specifies that it is possible to return to two different STATEs in the invoking

diagram E4. These two return points are specified by the R1 and R2 labels following the invoking of diagram E5. When the parser returns from diagram E5, it will either return to STATE S2 or STATE S4, depending on which return is specified. Thus, transition diagram E4 accepts both "ab" and "cd" as valid input strings, but does not accept either "ad" or "cb."

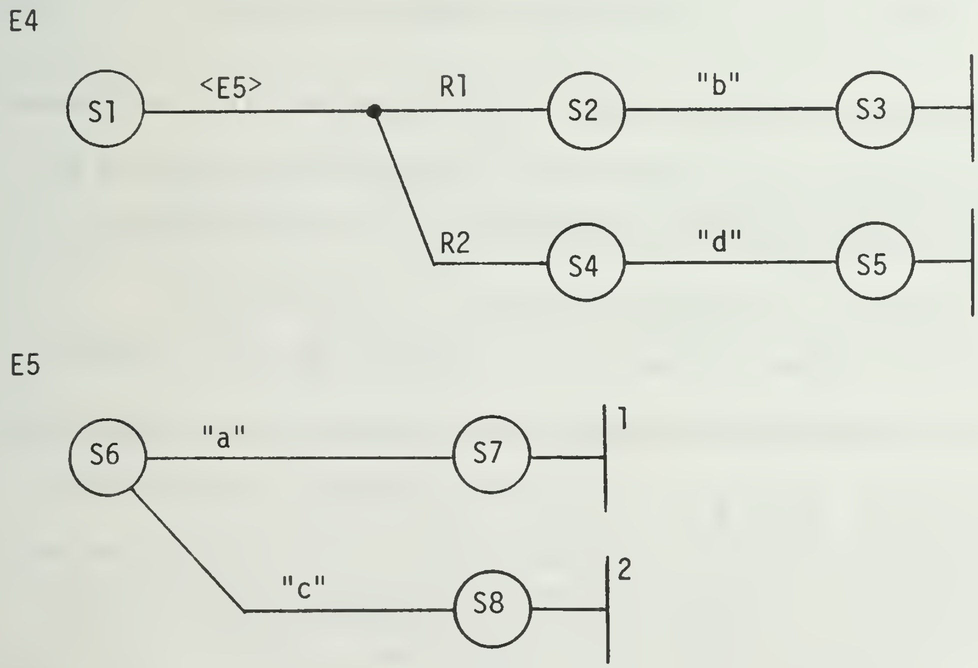


Figure 3.5 - Multiple return transition diagram example

A good example of where this multiple return feature is necessary arises in trying to parse a (simplified) PL/I conditional expression as shown in Figure 3.6.

simple expression parentheses

(((I + 100) * J) = K)

conditional expression parentheses

Figure 3.6 - PL/I conditional expression example

When the initial left parentheses "(" are examined, it is not known whether they are part of the overall conditional expression or are part of the inside simple expression on the left side of the relational operator. A transition diagram for a <conditional-expression> is shown in Figure 3.7. Note that the way in which this <conditional-expr> has been drawn corresponds to assuming that the initial "(" belongs to the overall conditional expression; if it turns out they actually belonged to the first simple expression, then the parse is resumed at point δ, which accepts the ")" and continues parsing the simple expression.

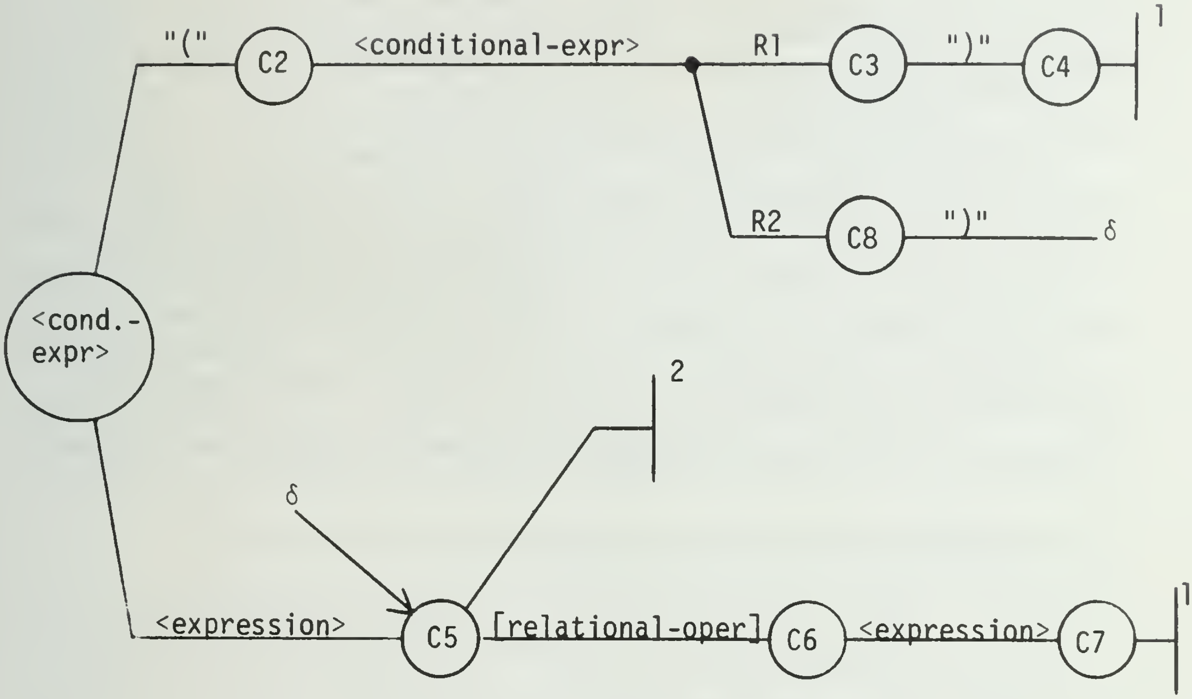


Figure 3.7 - PL/I <conditional-expression> transition diagram

3.3 Formal Description of the Transition Diagram Model

As a more formal description, a transition diagram system consists of a set of nested push-down automata (NPDA) that have the capability of invoking one another. Each NPDA is capable of reading a portion of the input string and accepting or rejecting it. Lomet [Lom., '73] calls the NPDA that are capable of being invoked "submachines"; the initial STATE of a submachine is known as its "entry" STATE and a submachine is invoked by the use of its entry STATE number by another NPDA. This results in the invoking STATE being saved at the top of a parser stack and the parse resumed at the new entry STATE. Each submachine also contains one or more

"exit" STATEs; when an exit STATE is reached, the top of the stack together with the particular exit STATE determine the STATE in the original invoking NPDA with which to continue the parse. An error in the parse is detected if a NPDA reads a token from the input string for which there is no corresponding STATE transition that the NPDA can make. The reader is referred to the discussion by Lomet [Lom., '73] for further technical details about transition diagrams and the class of languages accepted by transition diagrams.

3.4 Description of the Implemented Parser

A table-driven parser system has been designed that incorporates the actions described in the preceding sections of this chapter. The syntax table is actually an encoding of the transition diagrams into a small, interpretable instruction set. The syntax parser of the compiler consists of a routine that interprets this instruction set in an appropriate way. (This routine can be called the "table interpreter" or "table-driven" routine.)

The syntax parser has control over and maintains certain tables and data structures within the compiler. All of these tables and structures are located in a region of memory that is referred to as the parser storage area.

One particular variable that is maintained is the parser STATE variable. This STATE variable corresponds directly to the nodes or states that have been described for a transition diagram.

As each token is accepted by the parser, it is associated with the particular STATE that the parser was in. If a token must be "uncompiled," then the parser is reset to the associated STATE for that token, that is, as if the token has not yet been parsed. The value of this STATE variable always points to some instruction in the syntax table. As the input string is parsed, this STATE variable is updated (i.e., moved to point to a different instruction in the table) to reflect the current state of the parse.

The parser maintains a regular parsing return-state stack which is used when one transition diagram invokes another. Entries on this stack all point to the location in the parser table where the nested transition diagram was invoked. Any changes in the parser stack are recorded in the parser-action history for the token that is currently being parsed, allowing the stack to be properly reestablished when a token is "uncompiled."

The preceding discussion in this chapter describes a parser model that is sufficiently powerful to recognize all deterministic context-free languages in time proportional to the length of the input string [Lom., '73]. A few extensions to the model have actually been included in the compiler's parser system to enable the handling of certain context-sensitive semantic requirements such as proper and consistent declaration of attributes for an identifier, consistent references to declared array identifiers, etc. These extensions include allowing auxiliary memory variables

to be utilized by the parser (for operations such as counting the number of subscripts in an array reference), allowing the labels on any branch in a transition diagram to refer to any of these auxiliary variables or any symbol table field, and allowing special, language-dependent semantic-subroutines to be invoked at appropriate times during the operations of the parser system. Any changes that are made to any of these auxiliary variables or symbol table fields are recorded in the parser-action history for the token that is being parsed.

A complete description of the design and operations of the actual implemented parser can be found in [Tin., '75].

3.5 Compiler Environment when Parser Signals Error

As stated previously, a syntax error is detected by the parser if the attributes of the current input token do not match any of the labeled branches for a STATE. When this happens, the following environment is assumed by the error diagnostic system that is described in the remainder of this thesis.

The intermediate text generated by the compiler is very simply a tokenized version of the source text. The advantages of this are that only one copy of the program need be stored, "reverse compilation" or "uncompiling" can be implemented easily without complicated reversal of code generation, and the original program can be displayed at any time simply by examining the intermediate

text. As each token is output from the lexical analyzer, it is entered into the rightmost end of the intermediate text. Thus, if a syntax error is detected, the last token in the intermediate text is always the illegal one that the parser could not accept. In addition, no right context is available for examination; only the intermediate text entries up to and including the error token are available.

The parser-action history is a vector that is associated with the intermediate text. Each entry in the history records some change that occurred in the parser environment. A correlation is maintained between entries in the history and tokens in the intermediate text.

When a syntax error is detected, the current parser STATE is available, as well as the history of all STATE changes that have occurred for the previous tokens in the intermediate text. Similarly, the current parser return-state stack is available, as well as the history of all stack changes that have occurred. Finally, the current symbol table and the auxiliary parser storage variables that are used by the parser for the particular language are available; all changes to these data areas are recorded in the history.

This concludes the summary of the parser environment that the error analysis routines can work with.

Chapter 4

A MODEL FOR DIAGNOSTIC INTERACTION WITH THE PROGRAMMER

4.1 Goals for Diagnostic Interaction

This chapter describes a useful model of the interaction that can occur between a programmer who has attempted to enter a syntactically illegal token into the compiler and the error diagnostic system that is trying to describe the error to the programmer. Because the programmer is assumed to be relatively inexperienced at writing and debugging problems, the diagnostic system must be very simple to use. All of the actions taken by the diagnostic system must be transparent to the programmer. In particular, any messages that are displayed for the programmer must be simple to understand and must be stated using the terminology that the programmer is familiar with. Concepts such as "statements," "subscripts," "expressions," etc., which are important in the programming language and which are presumably recognized by the programmer should be referred to when appropriate by the diagnostic system.

In addition, the diagnostic system should make use of the fact that the programmer is at a timesharing terminal and thus can be used to control some aspects of the error analysis and recovery operations. In particular, the programmer should have some control over the level of detail that is given in diagnostic messages. If

a "numeric expression" is referred to in a message, for example, the programmer may or may not be interested in more detail about the form of a "numeric expression"; while the more detailed information should always be available, it should not always necessarily be presented.

Likewise, as opposed to most batch-oriented compilers, the interactive compiler's diagnostic system needs to make no attempt to actually "correct" the error or to modify the program and "recover" by itself. Since the programmer is directly available, the system should attempt to explain the error to him, requiring however that he be the one to actually back up, edit, and repair the program.

4.2 Suggesting "Possible Corrections" for the Programmer

To satisfy the goals for diagnostic interaction just mentioned, we propose a diagnostic system that interacts by presenting various suggestions of "possible corrections" of the program to the programmer. After a suggestion has been presented, the programmer then has control to either request that a different or more detailed suggestion be made by the system or else to return from the diagnostic system back to the editor system and proceed to modify the input and attempt to correct the error. This technique is very simple for the programmer to use since the only interaction that is required from him is that he examine the suggestions that

are made and either request other suggestions or request the return to the editor system.

The example shown in Figures 4.1(a) - (d) illustrates the types of suggestions that can be made by the diagnostic system. It is an example of a short PL/I program segment in which the illegal token "THEN" has been entered at the end of an assignment statement by the programmer. Each successive display (a) - (d) was requested by the programmer via the "HELP" key on the PLATO keyboard. In all cases, upon detecting the "HELP" request, the diagnostic system erased the current suggestion message that was being displayed and presented the next message to the programmer; the program itself was always in view on the screen, with only the diagnostic messages being changed.

```
FILE = workspace           PL/I WORKSPACE(14-new)        SPACE = 26Ø

.TEST1Ø: .PROC;
..........DECLARE.(I,J).FIXED;
.LP1:.....GET.LIST(I,J);
..........I.=.J...THEN

**********ERROR**********                       BACK or ERASE to fix.
This reserved word is not permitted here.
   Press HELP for more information.
```

Figure 4.1(a) - Example of suggestion messages

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 260

.TEST10: .PROC;
.........DECLARE.(I,J).FIXED;
.LP1:....GET.LIST(I,J);
.........I.=.J...[THEN]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with an arithmetic operator.
   Press (HELP) for a different suggestion.
```

Figure 4.1(b) - Example (continued)

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 260

.TEST10: .PROC;
.........DECLARE.(I,J).FIXED;
.LP1:....GET.LIST(I,J);
.........I.=.J...[THEN]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with ";".
   Press (HELP) for a different suggestion.
```

Figure 4.1(c) - Example (continued)

```
FILE = workspace           PL/I WORKSPACE(14-new)        SPACE = 26Ø

.TEST1Ø: .PROC;
.........DECLARE.(I,J).FIXED;
.LP1:....GET.LIST(I,J);
.........[I].=.J...THEN

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Insert a reserved word "IF" in front of [    ].
```

Figure 4.1(d) - Example (continued)

Note also that all of the available options for the programmer are displayed with each message: in this example, the "HELP" key is active for each message as well as the regular compiler editing "BACK" and "ERASE" keys which allow the programmer to modify and correct the program.

A further example of the control that the programmer has over the suggestions is given in Figures 4.2(a) - (e). In this PL/I example, the programmer has attempted to declare identifier "J" with conflicting attributes. (The "VARYING" attribute is associated with character strings in PL/I, not decimal variables.) The suggestion given in Figure 4.2(b) includes a different option for the programmer: if another suggestion is needed, the

programmer can either request to see more detail about "attributes" that are consistent with "DEC" (the detailed suggestions are shown in Figures 4.2(c) and (d)), or else can request that some other different suggestions be made (Figure 4.2(e)). Note also that if the programmer elects to see more detail, then following the suggestion in Figure 4.2(d) (or by requesting a different suggestion in Figure 4.2(c)), the next suggestion is the one shown in Figure 4.2(e); therefore the programmer will not miss seeing any suggestions because of electing to see more detail on an earlier suggestion.

Further examples are given in Appendix II.

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 276

.TEST.:..PROC;
.........DCL.I.FLOAT;
.........DCL..B.DEC.[VARYING]

**********ERROR**********.                    [BACK] or [ERASE] to fix.
 This attribute is not permitted here.
   Press [HELP] for more information.
```

Figure 4.2(a) - "Detail" suggestion messages example

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 276

.TEST.:..PROC;
.........DCL.I.FLOAT;
.........DCL..B.DEC.[VARYING]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an attribute.
   Press HELP for a different suggestion.
   Press SHIFT HELP to see a legal attribute.
```

Figure 4.2(b) - "Detail" example (continued)

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 276

.TEST.:..PROC;
.........DCL.I.FLOAT;
.........DCL..B.DEC.[VARYING]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an attribute "FLOAT".
   Press HELP for a different suggestion.
   Press SHIFT HELP to see another legal attribute.
```

Figure 4.2(c) - "Detail" example (continued)

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 276

.TEST.:..PROC;
.........DCL.I.FLOAT;
.........DCL..B.DEC.VARYING

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an attribute "FIXED".
   Press HELP for a different suggestion.
```

Figure 4.2(d) - "Detail" example (continued)

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 276

.TEST.:..PROC;
.........DCL.I.FLOAT;
.........DCL..B.DEC.VARYING

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with ",".
   Press HELP for a different suggestion.
```

Figure 4.2(e) - "Detail" example (continued)

4.3 Algorithmic Notation

Short algorithms will be presented throughout this thesis. The notation that will be used in stating these algorithms will be a readable, Algol-like language that contains whatever language constructs are necessary to most clearly illustrate the particular algorithm being presented. The left arrow "←" is the assignment operator. Typical control constructs include an "if - then - else" statement (which is terminated with a "fi" in all cases to avoid ambiguity), a looping construct (generally "loop. . .while. . .repeat"), and a procedural "begin - end" construct. Also, in many cases, certain operations in the algorithm will not be refined explicitly when the algorithm is given; these undetailed operations include both references to other procedures in the error analysis system (denoted with double slashes "//" around a description of the procedure) and also local operations that must be performed (denoted with single slashes "/" around a description of the effect of the local actions).

The notation used in these algorithms is not intended to be rigorously defined; the goal is for the operation and effect of the algorithm to be stated as clearly as possible.

4.4 Algorithm for Diagnostic Interaction

An algorithm that provides a suitable model for the diagnostic interaction described in this chapter is given as

Algorithm 'USER_INTERACTION' in Figure 4.3. This algorithm is purposely simple and nondetailed; Chapter 5 will more fully describe a detailed implementation of this algorithm.

```
-   begin  USER_INTERACTION
-        /initialize error system environment/
-        //display:  signal error to user//
-        loop:
-             //prepare new suggestion message//
              comment  "Read-edit-key" waits until an edit key or
                       the "HELP" key is pressed; the key-type is
                       returned in "Read-key."
-             /Read-edit-key/
-        while  Read-key = "HELP":
-             //erase:  current message on screen//
-             //display:  new message on screen//
-        repeat:
-        /reestablish program editing environment/
-        /return to program editing mode/
-   end  USER_INTERACTION
```

Figure 4.3 - Algorithm 'USER_INTERACTION'

Chapter 5

DETERMINING "POSSIBLE CORRECTION" SUGGESTIONS

5.1 Introduction

The largest problem to be solved in algorithm "USER_INTERACTION" (shown in Figure 4.3) is that of determining what "possible correction" suggestions can be made to the programmer as diagnostic help is requested. This chapter first describes a preliminary model of an algorithm for determining suggestions that is similar to the algorithm suggested by Levy [Lev., '71]. Then a more sophisticated algorithmic model is given and shown to produce more comprehensive suggestions than the preliminary model.

5.2 A First Model for Determining "Possible Correction" Suggestions

5.2.1 J. P. Levy

A thesis by J. P. Levy [Lev., '71] defined and studied a number of problems associated with automatic error correction techniques. If the "error token" is defined to be the input token at which a syntactic error is detected, then the "actual syntactic error" that has occurred is not necessarily located at the error token, but may in fact be located someplace to the left of the error token. Levy proved (Theorem, section III.3) that the actual error may not even be located "close" to the error token, but could have occurred an unbounded distance to the left of that token.

However, for any particular error situation, Levy showed it is possible to find a location in the input string such that the actual syntactic error could not have occurred before that location: this location is defined as the "left context" of the error. (Locating this place in the input string, which is called a "beacon," is a difficult problem and Levy devotes much of his thesis to its solution.) The fact that such a "left context" location exists is trivially true, since the extreme limit for its location is the beginning of the input string; however, Levy shows that it is possible for the left context location to be located much closer to the error token than the beginning of the input string in many situations.

Then Levy defines "error interpretations" of the input string to be syntactically correct strings that are similar to the original input string, but contain a few changes or "corrections"; if the number of changes made to the original string to arrive at an interpretation is K, then the original string is said to have K errors. He shows that all interpretations with K changes will become "equivalent" at some location to the right of the error point if it is possible to interpret the string correctly with only K changes; this equivalence means that the parser has settled into a common state for all of the interpretations at that point, and thus the parse for the rest of the string is similar for all interpretations.

Levy defines three operations that are used in building the error interpretations: "R," "A," and "S." The "R" operation specifies that, at each point in the input string, alternate parses (interpretations) should be created by replacing the actual input token with each of the tokens that would be syntactically acceptable at that point. The "A" operation specifies that, at each point, alternate parses should be created by adding to the input string each of the tokens that would be syntactically acceptable. Finally, the "S" operation specifies that, at each point, an alternate parse should be created by suppressing from the input string the actual input token that was there.

These three operations can be conveniently expressed as follows: if the input string at some point is expressed as

$$\times\alpha$$

where $\times$ is the first token not read by the parser and α is a string containing the remainder of the input tokens, then if t is an acceptable token at this point, the "R" operation changes the input to

$$R_t\ (\times\alpha) = t\alpha,$$

the "A" operation changes the input to

$$A_t\ (\times\alpha) = t\times\alpha,$$

and the "S" operation changes the input to

$$S_t\ (\times\alpha) = \alpha.$$

Levy proves that all syntax errors can be corrected by some combination of these three operations.

Finally, Levy describes an error correction algorithm that operates as follows: following the detection of an error, back up in the input string until the left context location is reached; then reparse the input string from this point forward creating all possible modified interpretations of the input string with N or fewer changes (for some fixed N) until all of the interpretations are equivalent, that is, until the right context for the error is found; then select one of the interpretations with the smallest number of changes as the "correction" for the program.

5.2.2 Modifications to Levy's Analysis

The algorithm described by Levy can be modified to make a good preliminary model for determining "correction" suggestions when used with a transition diagram parsing system. However, a number of important changes must be made to apply Levy's basic ideas in an interactive environment. For one thing, in Levy's system, all of the interpretations had to be initially computed for the error situation, and then a "best" interpretation selected and used to correct the program. However, in an interactive environment, it is not necessary for the error system to guess which error interpretation is the "best" interpretation; the interpretations can be given successively to the programmer, and the programmer can

then select the appropriate interpretation and use it to correct the program. In this way, the error system is acting not so much like an automatic error correction and recovery system as like an automatic error "consultant." When an error is detected by the compiler, the error system then provides the programmer with the different possibilities for correcting the error.

Since the different error interpretations are going to be given successively to the programmer, it is not necessary for all of the interpretations to be computed immediately upon the detection of an error by the compiler. In an interactive environment, it is important to try to spread the computing load out over a period of time if possible, and to avoid heavy bursts of calculations in short periods of time. Also, since the programmer will frequently examine more than one error interpretation for an error situation, it is desirable that the suggested corrections to the program be given to the programmer in an organized order.

It is proposed that instead of the error system initially moving back in the input string to the left context location and then starting the computations of the error interpretations, the error system should compute the interpretations in the reverse order. Initially, the error system will compute all error interpretations assuming that the entire input string up to the error token is correct and the error token itself is in error. As different interpretations are found, they are presented to the

programmer as in algorithm "USER_INTERACTION" (Figure 4.3). When no more interpretations can be found from this input location, then all interpretations (if any) will be computed from one token farther back in the input string, and so on. At each of these new locations the algorithm makes the assumption that the error token itself is correct and that the "actual syntax error" occurred someplace back in the input string.

The backup process should terminate when a token is reached in the input string where it is certain that the error actually occurred after that token. Levy defines this left context location as a "beacon," and he thoroughly discusses the problems associated with locating the beacon for a given error situation. The reader is urged to consult this work for the details; this current thesis simply assumes that such a "beacon" location can be found.

This algorithm is intuitively good from the programmer's point of view, since the more "local" corrections are suggested first, followed later by more "global" suggestions. From a computational point of view, this algorithm is also by far the best approach. It is very easy to compute the initial error interpretations, but much more time-consuming to calculate interpretations with modifications that occur far back in the input string. The hope is that usually the initial interpretations and suggestions will satisfy the programmer and, while always available, only occasionally will more complex interpretations need to be determined.

Finally, note one more major departure from Levy's system. In the interactive compiling environment that was described in the previous chapters, there is no right context available when an error is detected and analyzed. Therefore, although the input context to the left of the error token is still available and important, there is <u>no</u> context to the right of the error token in the input string. Because of this, the error interpretations as described by Levy are "equivalent" in the interactive environment when all of the input string up to and including the error token has been considered. Another way of describing this is to say that the error system can compute error interpretations of the input string that just consider the left context and the actual error token; it is then the responsibility of the programmer to (mentally) supply the intended right context (that has not yet been entered into the program) to decide if a particular error interpretation constitutes the appropriate correction to the program. Thus, the overall diagnostic system in an interactive environment requires the joint participation of both the analysis system's interpretation routine and the analysis of the programmer to satisfactorily diagnose the syntax error.

5.2.3 Token Option Lists

Before the preliminary model of the "suggestion builder" routine can be given, one more important routine must be mentioned.

It is possible, for each state in a transition diagram, to construct a list containing all of the tokens that are acceptable to the parser if it is in that state. This list corresponds to what Griffiths [Gri., '74] has called a "director symbol set" in LL(k) terminology; it will be called the "token" option list in this thesis, and the routine that builds the option list for a state will be called the OPT routine.

It is very important to realize that the acceptable token list for a state depends not only on just the transition diagrams themselves, but also on a particular complete parser environment; the token list for a state will vary depending on other elements of the parser's environment. If all of the branches in the transition diagram that leave state S are labelled with just token specifications, then those tokens comprise the entire list for state S. However, if one of the branches invokes another transition diagram (with entry state Q), then any simple tokens that are specified for the original state, S, must be added to the list, and then the entry state Q for the invoked transition diagram must be examined as above. Likewise, if state S is a final state for some transition diagram, then any simple tokens that are specified for state S must be added to the list, and then the current transition diagram exited according to the parser's stack at that time. Eventually, all of the possible tokens will be resolved and added to the list, since the parser accepts only actual tokens in the input string.

```
-    begin  transit (T, S)
     comment  T is an input token, and S is a parser state.
     comment  transit is a function that returns a value.
              The value is a new state number if a parser
              state transition from STATE S with token T can be
              made.
              Otherwise the value is "null."
     comment  The function is not detailed here since the explicit
              actions depend directly upon the compiler's imple-
              mentation.
-    end  transit
```

Figure 5.1 - The "transit" function

5.2.4 The Preliminary Model

Given that this token option list can be determined for any state, the preliminary algorithm consists of three routines: "MODIFY," "REPARSE," and "USER_INTERACTION." MODIFY will make a modification to the input string and invoke REPARSE to determine whether the corrected string is then acceptable. If so, REPARSE will invoke USER_INTERACTION; otherwise it will recursively invoke MODIFY to make further modifications if the current interpretation is still not acceptable.

Both MODIFY and REPARSE use a function "transit." A

description of the effect of the transit function is given in Figure 5.1. The function accepts two parameters:

T, an input token, and

S, a state of the parser.

The function checks to see if a legal parser state transition can be made from STATE S with the token T. If so, then transit returns the value of the new state number; otherwise transit returns a "null" value.

The MODIFY routine has three input parameters:

L, a pointer to some location in the input string (indexed from the beginning of the string),

Q, the corresponding parser state that is expecting the token at location L, and

K, a counter indicating the number of changes that have currently been made to the input string.

When the MODIFY routine is activated, first the token option list for state Q is constructed. Then, for each token option in the list, two modifications can be made: Add the token option to the input string in front of the token at location L, or Replace the token at location L with the token option. After the above modifications have been tried for each token option, a final modification consisting of Suppressing the actual token at location L from the input string can be tried.

Further describing these operations, if the MODIFY

routine tries an Add operation, it will try to add the token option to the input string in front of the token at location L; if the original input string is $x\alpha$, and the token option is t, then the modified input string is $tx\alpha$, starting in parser state Q. However, this can be viewed with the same effect as the modified input string $x\alpha$ starting in parser state Q_t, where Q_t is the state of the parser following the parsing of token t when in state Q (that is, Q_t = transit (t, Q)). Said differently, the Add operation is

$$A_t\left(\underset{(Q)}{\;}x\alpha\right) = \underset{(Q)}{\;}tx\alpha = \underset{(Q_t)}{\;}x\alpha,$$

where the lower line indicates the state of the parser at that location in the input string.

Similarly, the Replace operation can be specified as

$$R_t\left(\underset{(Q)}{\;}x\alpha\right) = \underset{(Q)}{\;}t\alpha = \underset{(Q_t)}{\;}\alpha,$$

and the Suppress operation can be specified as

$$S_t\left(\underset{(Q)}{\;}x\alpha\right) = \underset{(Q)}{\;}\alpha.$$

For each of the modifications described above, the remainder of the modified input string must then be reparsed to see if it now comprises a correct, syntactically legal string. The reparsing controller, called REPARSE, has the same formal input parameter list as does the MODIFY routine:

LR, a pointer to some location in the input string (indexed from the beginning of the string),

QR, the corresponding parser state that is expecting the token at location LR, and

KR, a counter indicating the number of changes that have currently been made to the input string.

REPARSE is a highly self-recursive routine. When invoked, the routine first checks to see if the current LR input location has reached the end of the input string. If so, then the reparse is successful and a correct interpretation has been found. Whatever modifications were made to the input string can be used as a "possible correction" suggestion by the error system routine "USER_INTERACTION," since making those corrections to the input string yields a syntactically correct interpretation of the input.

If the input is not yet exhausted for the current invoking of REPARSE, then the routine will first extend the current reparse attempt by one more token if the token at location LR is acceptable to the parser from STATE QR. The extension operation consists of recursively invoking the REPARSE routine with the input pointer and corresponding parser state advanced by one token.

Finally, the REPARSE routine will recursively invoke the MODIFY routine at the current input location if the maximum number of modifications have not already been made in the current reparse attempt. The MODIFY routine will then make some modifications at the new location in the input and again initiate the reparsing.

The effect of the operation of the MODIFY and REPARSE routines is to construct all possible alternate legal interpretations of the input string with N or fewer modifications. Furthermore, since the REPARSE routine tries to extend the current reparse attempt before forcing another modification to be made, legal interpretations first are found with modifications closest to the original error token location ("local" modifications), with later legal interpretations containing modifications farther away from the error token ("global" modifications).

Algorithms for the token MODIFY and the REPARSE routines are given in Figures 5.2 and 5.3, respectively. In the REPARSE algorithm, "error_token_location" refers to the location of the original token that caused the parser to signal an error; this location is the location of the rightmost token in the input string. Also in the REPARSE algorithm, the action "success" activates the "USER_INTERACTION" algorithm with a "possible correction" suggestion; for clarity, this linkage is not given in Figure 5.3.

In the MODIFY algorithm, notice the three different references to the REPARSE algorithm. In all three, the change count field KR is incremented by 1 to indicate that a new modification has been made. The specification of a REPARSE LR location of L + 1 indicates that the reparse attempt is to begin with the next token in the input string following the one at location L. This corresponds to the preceding discussion on the effect of the Add, Replace, and Suppress operations.

```
-    begin  MODIFY (L, Q, K)
     comment  L is the input location pointer (indexed from the
                beginning of the input string),
              Q is the parser state for token at location L,
              K is number of modifications that have been made
                prior to current activation.
-    //  OPT : build token_option_list for state Q//
-    for all tε token_option_list loop :
-         // REPARSE (L, transit (t, Q), K + 1)// comment Add
               operation.
-         // REPARSE (L + 1, transit (t, Q), K + 1)// comment
               Replace operation.
-    repeat
-    // REPARSE (L + 1, Q, K + 1) // comment Suppress operation.
-    end  MODIFY
```

Figure 5.2 - "Token" MODIFY routine

```
-   begin  REPARSE (LR, QR, KR)
    comment  LR is reparse start input location index,
             QR is reparse start state for location LR,
             KR is number of modifications made to input string.
    comment  Is reparse attempt successful?
-   if LR > error_token_location then "success";
-   else
         comment  Extend reparse attempt 1 token if next token
                  is acceptable.
-        if transit (token (LR), QR) ≠ "null" then
-             //REPARSE (LR + 1, transit (token (LR), QR), KR)//
-        fi
         comment  Make another modification at location LR, if
              possible, to build all legal interpretations.
-        if  KR < N then
-             //MODIFY (LR, QR, KR)//
-        fi
-   fi
-   end  REPARSE
```

Figure 5.3 - REPARSE routine

5.2.5 Examples using the Preliminary Model

As an example of the analysis performed by the preliminary model that has been presented, consider the transition diagram E1

shown in Figure 3.1. For transition diagram E1, Table 5.1 shows the token options that can be computed for each of the different states.

A few specific unacceptable input strings, along with some possible interpretations for these strings, are shown in Table 5.2. This table gives the original token input string and then a number of different "corrected" strings; for each "corrected" string interpretation, the number of modifications required to obtain the error interpretation is given, as well as a sample "suggestion" message that could be used to describe the modifications that were made.

STATE	TOKEN OPTIONS
S1	"a," "b"
S2	"b," "d"
S3	"c"
S4	"d"
S5	"e"
S6	none
S7	"f"

Table 5.1 - Token Options for States of Transition Diagram E1

TOKEN INPUT STRING	ERROR INTERPRETATION	NUMBER OF CHANGES MADE	SAMPLE "SUGGESTION" MESSAGE
"a d c"	"a d e"	1	Replace "c" with "e."
	"a b c"	1	Replace "d" with "b."
"a b f"	"a b c"	1	Replace "f" with "c."
	"b f"	1	Remove "a" from the input.
"b c"	"b f"	1	Replace "c" with "f."
	"a b c"	1	Insert "a" in front of "b."
"b f d"	"b f e"	1	Replace "d" with "e."
	"a b c d"	2	Insert "a" in front of "b" and Replace "f" with "c."
"a e"	"a b"	1	Replace "e" with "b."
	"a b c d e"	3	Insert "b," "c," and "d" in front of "e."
	"b f e"	2	Replace "a" with "b" and Insert "f" in front of "e."

Table 5.2 - Error Interpretations for Transition Diagram E1

STATE	TOKEN OPTIONS
S1	"a," "b"
S2	"f," "b"
S3	"d"
S4	"e"
S5	"c"
S6	"f," "b"
S7	"g"
S8	"c"
S9	"d"
S10	none

Table 5.3 - Token Options for States of Transition Diagrams E2 and E3

As a second example of the analysis performed by the diagnostic model, consider the more complex transition diagrams E2 and E3 shown in Figure 3.3. Table 5.3 shows the different token options that can be computed for each state in the diagrams; notice that to obtain the options for state S2 requires examining the token options for the entry state for diagram E3, state S6, and similarly to obtain the options for state S9 requires returning from diagram E3 back to state S3, and then examining the token options for state S3.

A few unacceptable input strings for the diagrams E2 and E3, along with some possible error interpretations for these strings, are shown in Table 5.4.

TOKEN INPUT STRING	ERROR INTERPRETATION	NUMBER OF CHANGES MADE	SAMPLE "SUGGESTION" MESSAGE
"a d"	"a f"	1	Replace "d" with "f."
	"a b"	1	Replace "d" with "b."
	"a f g d"	2	Insert "f" and "g" in front of "d."
"a c"	"a b c"	1	Insert "b" in front of "c."
	"b c"	1	Replace "a" with "b."
"b c d"	"b c e"	1	Replace "d" with "e."
	"a b c d"	1	Insert "a" in front of "b."
"a b c e"	"a b c d"	1	Replace "e" with "d."
	"b c e"	1	Remove "a" from the input.
"b e"	"b c e"	1	Insert "c" in front of "e."
	"a b c d e"	3	Insert "a" in front of "b" and Insert "c" and "d" in front of "e."

Table 5.4 - Error Interpretations for Transition Diagrams E2 and E3

5.3 Extended Suggestion Model

5.3.1 Using Nonterminal Information

One of the objectives of an interactive diagnostic system is to provide for the programmer as much information about the state of the parse and the internal state of the compiler as is possible. The preliminary suggestion model presented earlier in this chapter (Figures 5.1 and 5.2) is able to provide much information about modifying individual tokens in the input string to correct the error. However, no information is available in the preliminary diagnostic model about higher-level language features such as "expressions," "subscripts," "statements," etc., even though the transition diagram parser system that is being used in the compiler provides and uses this information.

In Chapter 3 of this thesis, the action of a transition diagram parser is described as making a state transition from a current state to some next state, depending on the input token and the labeled branches for the current state. A regular parser stack is used to allow the invoking of "sub" transition diagrams when specified on a branch; this operation corresponds to attempting to parse a "nonterminal" in the programming language, and these nonterminals typically represent higher-level language constructs in the language.

Therefore, for any state in the parser, it is possible

to determine not only what actual tokens are acceptable parsing options, but also what nonterminals, if any, are being sought. It is possible, for any state, to build a list containing all token options and all intermediate nonterminals that are acceptable to the parser. The algorithm for this is almost the same as the algorithm for the token option builder that was described in section 5.2.3; the only difference is that when a branch for a state refers to a nonterminal, then first that actual nonterminal is added to the list, and then the nonterminal's entry state is examined. This new list will be called the general option list.

It is also possible to view the input string either as the string of actual tokens that have been input, or else as a string of tokens and nonterminals that have been parsed. Using the example E3 given in Figure 3.3, the input string "abcd" can be viewed either as

"a b c d"

or as

"a E3 d."

To use these new ways of viewing the input string and the requirements of the parser for any state, it is necessary to change the activities of the MODIFY routine. In the preliminary version of MODIFY the idea was, for each token option in the token option list for a state, to create alternate interpretations of the input by applying the "A" and the "R" operations, and

subsequently the "S" operation, with respect to the next token in the input string. In the new version, we want to create alternate interpretations of the input by applying, for each general option in the general option list, the "A" and the "R" operations, and subsequently the "S" operation, with respect to both the next token and the next nonterminal (if any) in the input string.

More formally, if the input string is

$$\times\ \beta\ \alpha$$

and the nonterminal B is

$$B = \times\ \beta,$$

then the input can be viewed as

$$\times\ \beta\ \alpha \text{ or } B\ \alpha, \text{ for state } Q.$$

The "A" operation, for some general option, g, changes the input to

$$A_g\ ({}_{(Q)}\ \times\ \beta\ \alpha) = {}_{(Q)}\ g \times \beta\ \alpha$$

$$= {}_{(Q_g)}\ \times\ \beta\ \alpha,$$

or

$$A_g\ ({}_{(Q)}\ B\ \alpha) = {}_{(Q)}\ gB\ \alpha = {}_{(Q_g)}\ B\ \alpha,$$

where Q_g is the parser state following the parsing of option g from state Q. This operation, then, specifies that a general option (that is, some acceptable token or some acceptable nonterminal) be added to the input string in front of the remainder of the

string. Note that for the "A" operation, it makes no difference how the remainder of the input is viewed, since the operation simply inserts an option in front of it.

For the "R" and "S" operations, however, viewing the input string as a simple string of tokens as opposed to a string of tokens and nonterminals makes a big difference. The "R" operation, for some general option, g, changes the input to either

$$R_g\,({}_{(Q)}x\ \beta\ \alpha) = {}_{(Q)}g\ \beta\ \alpha$$

$$= {}_{(Q_g)}\beta\ \alpha,$$

or

$$R_g\,({}_{(Q)}B\ \alpha) = {}_{(Q)}g\ \alpha = {}_{(Q_g)}\alpha,$$

depending on how the input is viewed. This operation implies the replacement of both the token and the nonterminal (if any) in the input string, with each general option for state Q (which may itself be either a <u>token</u> option or a <u>nonterminal</u> option); thus, all replacement combinations of tokens and nonterminals are performed.

Similarly, the "S" operation changes the input to either

$$S_g\,({}_{(Q)}x\ \beta\ \alpha) = {}_{(Q)}\beta\ \alpha,$$

or

$$S_g\left({}_{(Q)}B\,\alpha\right) = {}_{(Q)}\alpha,$$

depending on how the input is viewed. This operation implies the suppression of both the token and the nonterminal (if any) in the input string.

The second version of the MODIFY routine is given in Figure 5.4; this version incorporates the general option concept. The REPARSE routine is the same for this version of MODIFY as

```
-   begin  MODIFY (L, Q, K)
    comment  L is the input location pointer (indexed from the
               beginning of the input string),
             Q is the parser state number for token at location L,
             K is number of modifications that have been made
               prior to current activation.
-   //OPT : build general _option_list for state Q//
-   /Lnt ← location in input following the nonterminal that
         begins with token at location L; if no nonterminal begins
         there, set Lnt ← 0/
-   for all gε general_option_list  loop :
         comment REPARSE routine's 4th parameter is set to
             1)  'unsuccessful' : if previous modification did
                  not completely "correct" input string,
```

```
                 2)  'help' : if previous modification "corrected"
                              input string, but programmer requested
                              another suggestion
                 upon returning to MODIFY.
             comment Try Add operation.
-            //REPARSE (L, transit (g, Q), K + 1, last_try)//
             comment Prune special case for Replace operation.
-            if L ≠ error_token_location or last_try = 'unsuccessful'
                 then
                 comment Token Replace operation.
-                //REPARSE (L + 1, transit (g, Q), K + 1, last_try)//
                 comment Does nonterminal exist?
-                if Lnt ≠ 0  then
                     comment Nonterminal Replace operation.
-                    //REPARSE (Lnt, transit (g, Q), K + 1,
                         last_try)//
-                fi
-            fi
-        repeat
         comment Prune special case for Suppress operation.
-        if L ≠ error_token_location  then
             comment Token Suppress operation.
-            //REPARSE (L + 1, Q, K + 1, last_try)//
             comment Does nonterminal exist?
```

```
-         if  Lnt ≠ 0  then
              comment Nonterminal Suppress operation.
 -            //REPARSE (Lnt, Q, K + 1, last_try)//
-         fi
-    fi
-    end  MODIFY
```

Figure 5.4 - MODIFY routine, version 2

it was in Figure 5.3, except that it returns whether or not the reparse attempt was successful. Also, the transit function now accepts a general option as its first parameter, instead of just a simple token option.

Two other changes have been made to this MODIFY routine; both of these changes have to do with special cases of applying the "A," "R," and "S" operations. The "S" operation restriction is very easy to understand. Since no right context is available when an error is originally detected by the parser, it is true that a very trivial (though "possible") way of correcting the program is to suppress the last token that was input; however, this does not provide any more information for the programmer than was already known--that is, the last input token is unacceptable. To avoid this trivial suggestion, the "S" operation is not applied if the input location L (where the correction is to be made by the MODIFY routine) is the same as the location of the actual original error

token. This restriction prevents the diagnostic system from "finding" this "possible correction."

The second change has to do with the "A" and the "R" operations. Assume that the input location L is the same as that of the actual original error token. Then note that if an "A" operation is successful (that is, if some option can be successfully added to the input string in front of the error token), then it is also true that the "R" operation will succeed (replacing the error token with the option will yield a successful interpretation); this is again because of no right context being available for analysis. Therefore, to avoid nearly duplicate suggestions from being computed by MODIFY, if the "A" operation is successful for some general option with respect to the error token, then the "R" operation for that general option is not performed.

In summary, both of these special cases occur because of no right context being available in the input string. This forces slightly different considerations to be given to any changes that are attempted with respect to the location of the actual original error token in the input string.

5.3.2 Providing Programmer Control of Messages

A typical technique used to construct the <u>general option list</u> for a given parser state is to use a normal depth-first backtracking algorithm. This algorithm will begin by starting to examine each labeled branch for the original parser state. If the

label on a branch is a simple token specification, then that token can be added to the general option list, and the next branch examined. However, if the label is a nonterminal specification, then first that nonterminal is added to the general option list and then the branches for the entry state of the nonterminal are each examined in a similar fashion. When all of the branches for a particular state have been examined, the algorithm must backtrack to the state and branch where that entry state was invoked. The algorithm terminates when all of the branches for the original state have been examined.

If this depth-first technique is used, it is possible to provide some organization for the entries in the general option list. In particular, it is possible to view the general option list as a tree structure; the root node of the tree represents the original parser state Q; each son of the root node represents a different parser option for the state Q; finally, each of these sons that represent a nonterminal option as opposed to a simple token option is the root of a subtree that contains the further refining detail for that higher-level option. All of the leaves of the tree are simple token option nodes and all of the internal nodes in the tree are nonterminal nodes. This tree will be called the general option tree.

Using this general option tree, the MODIFY routine has a choice about which general option to use for its next loop

iteration: it can either proceed breadth first (across the initial sons of each node in the general option tree), or depth first (exhausting each son and corresponding subtree in the general option tree before proceeding to the following son). To avoid confusing the programmer, it seems intuitively best to use the depth-first order of selecting the next option. This has the effect of providing suggestions initially about a higher-level construct (nonterminal), with successive suggestions providing more and more detail about the high-level construct, until the detail reaches the actual token level. Note that since each node in the tree is actually a particular parsing option (all that the tree structure does is relate some of the options together), even if a modification attempt fails using a nonterminal option at a node high in the tree, an attempt with a descendent of that node may succeed; it is necessary to examine all the nodes in the tree to determine all possible corrections to the program. When all of the detail farther down the tree has been examined for one option (son) of a node, then another, different option (son) can be selected for that node and the depth-first examination resumed. When all of the options have been examined for a node, then the control moves one node farther back up the tree and the process is repeated.

With the general option tree structured in this way, it is possible for the programmer to control the amount of detail that is presented in some of the suggestion messages. If a message is

presented to the programmer that suggests that Adding or Replacing something in the program with a nonterminal will yield a correct interpretation, then the programmer can request that more detail be presented about the same suggestion message (if available); on the other hand, the programmer may indicate that no more detail is wanted about that suggestion message, but instead some other unrelated suggestion should be presented (if possible). As long as the programmer requests the more detailed suggestion messages, the MODIFY routine can just proceed with trying general options in the normal depth-first order. However, if a request for a "different" suggestion message is made by the programmer and the leaf (token) node has not been reached for the current path, then the depth-first order must be abandoned, and instead a new option (son) for the current node examined (i.e., move breadth-first instead of depth-first).

To handle the depth-first tree examination properly, the MODIFY routine has been divided into two routines: the new MODIFY routine creates the general option tree, invokes the new MOD_TREE routine to try the Add and the Replace operations in a depth-first order for the general option tree, and finally tries the Suppress operations. The MOD_TREE routine will move depth-first through the option tree unless the programmer requests that the detail about the current suggestion be restricted.

The final forms for the USER_INTERACTION, REPARSE, MODIFY and MOD_TREE routines are given in Figures 5.5, 5.6, 5.7, and 5.8,

respectively. These algorithms incorporate the changes to provide the programmer with control over the amount of detail that is presented in successive suggestions.

```
-    begin  USER_INTERACTION (request)
     comment  request is an output parameter.  It is set to either
          'successful-provide detail' or 'successful-restrict
          detail,' depending upon the programmer's keypress
          response.
     comment  USER_INTERACTION displays a new suggestion message on
          the screen and then waits for the programmer to respond.
          If the programmer requests another suggestion message,
          then "request" is set and the routine returns normally.
          If the programmer responds with any other compiler edit-
          ing key, then this routine forces the termination of the
          error analysis process, and exits directly into the com-
          piler's editor system.
     comment  Display new message.
-    //erase : current message on screen//
-    //display : new message on screen//
     comment  "Read-edit-key" waits until an editing key or one of
          the "help request" keys is pressed; the key-type is
          returned in "Read-key."
-    /Read-edit-key/
-    if  Read-key = Compiler-editing-key  then
-         /reestablish program editing environment/
```

```
        /exit from error system to the compiler's editor/
-   fi
    comment  The programmer has requested another suggestion.
        The normal request is the 'HELP' key, which requests that
        more detail be provided (if possible).  The alternate
        request is to restrict detail in the next message.
-   request ← if  Read-key = 'HELP' then 'successful-provide
        detail'
-       else 'successful-restrict detail'  fi
-   end  USER_INTERACTION
```

Figure 5.5 - Final USER_INTERACTION routine

```
-   begin  REPARSE (LR, QR, KR, user_request)
    comment  LR is reparse start input location index,
             QR is reparse start state for location LR,
             KR is number of modifications made to input string,
             user_request is an output parameter that specifies
                 whether the reparsing is successful or not.
-   /user_request ← 'unsuccessful'/
    comment  error_token_location is input location of the
        originally detected error token.
    comment  Is reparse attempt successful?
-   if  LR > error_token_location  then
-       if  KR = N  then
            comment  Reparsing was successful.  Invoke routine
                     USER_INTERACTION, which will either return
```

```
                              with user_request set indicating the amount
                              of detail the programmer wants, or else
                              will cause the error system process to be
                              terminated without returning normally.
-                    //USER_INTERACTION (user_request)//
-               fi
-          else
                comment  Extend reparse attempt 1 token if next token
                         is acceptable.
-               if  transit (token (LR), QR) ≠ "null" then
-                    //REPARSE (LR + 1, transit (token (LR), QR), KR,
                                                  user_request)//
-               fi
                comment  Make another modification at location LR, if
                         possible, to build all legal interpretations.
-               if  KR < N  then
-                    //MODIFY (LR, QR, KR)//
-               fi
-          fi
-          end  REPARSE
```

Figure 5.6 - Final REPARSE routine

```
-   begin  MODIFY (L, S, K)
    comment  L is the input location pointer (indexed from the
               beginning of the input string),
             Q is the parser state number for token at location L,
             K is the number of modifications that have been made
               prior to current activation.
    comment  OPT routine constructs the general_option_tree.
         It is a tree structure, with descendents of a node repre-
         senting further refinements or detail for the node.
-   //OPT:  build general_option_tree for state Q with root
         'root'//
-   /Lnt ← location in input following the nonterminal that
         begins with token at location L; if no nonterminal
         begins there, set Lnt ← 0/
    comment  Try Add, Replace operations for sons of root of tree.
-   //MOD_TREE (L, Q, K, Lnt, root)//
    comment  Prune special case for Suppress operation.
-   if L ≠ error_token_location  then
         comment  Token Suppress operation.
-        //REPARSE (L + 1, Q, K + 1, last_try)//
         comment  Does nonterminal exist?
-        if  Lnt ≠ 0  then
              comment  Nonterminal Suppress operation.
-             //REPARSE (Lnt, Q, K + 1, last_try)//
```

```
-           fi
-      fi
-      end  MODIFY
```

Figure 5.7 - Final MODIFY routine

```
-      begin  MOD_TREE (LD, QD, KD, LntD, Droot)
       comment  LD is the input location pointer,
                QD is the parser state number for token at location
                   LD,
                KD is number of previous modifications,
                LntD is location in input following nonterminal or
                     0 if no nonterminal exists,
                Droot is the root of the (sub) tree in general_
                      option_tree.
       comment  MOD_TREE tries Add and Replace (token and nonterminal)
                operations for each son (option) of Droot.
       comment  REPARSE routine's 4th parameter is set to
                1)  'unsuccessful' : if previous modification did
                         not completely "correct" input string, or
                2)  'successful-provide detail' or
                    'successful-restrict detail' : if previous
                         modification "corrected" input string, but
                         programmer requested another suggestion.
-      for all  g ε sons (Droot)  loop :
```

```
comment  Add operation.
//REPARSE (LD, transit (g, QD), KD + 1, last_try)//
comment  Is more detail requested?
if  last_try = 'successful-provide detail'  then
    comment  Give more detail:  move depth-first through
             tree.
    //MOD_TREE (LD, QD, KD, LntD, g)//
fi
comment  Prune special case for Replace operation.
if (LD ≠ error_token_location or last_try = 'unsuccess-
    ful') and last_try ≠ 'successful-restrict detail'
    then
    comment  Token Replace operation.
    //REPARSE (LD + 1, transit (g, QD), KD + 1,
        last_try)//
    comment  is more detail requested?
    if last_try = 'successful-provide detail' then
        comment  Give more detail:  move depth-first
                 through tree.
        //MOD_TREE (LD, QD, KD, LntD, g)//
    fi
    comment  Does nonterminal exist?
```

```
-               if  LntD ≠ 0 and last_try ≠ 'successful-restrict
                    detail'
                     then
                     comment  Nonterminal Replace operation.
-                    //REPARSE (LntD, transit (g, QD), KD + 1,
                          last_try)//
                     comment  Is more detail requested?
-                    if last_try = 'successful-provide detail' then
                          comment  Give more detail:  move depth-
                                   first through tree.
-                         //MOD_TREE (LD, QD, KD, LntD, g)//
-                    fi
-               fi
-          fi
-     repeat
-     end  MOD_TREE
```

Figure 5.8 - Final MOD_TREE routine

5.3.3 Example using the Extended Model

As a comparative example of the error interpretations that can be determined using the Extended Diagnostic Model that has been described rather than the preliminary Token Model, consider again the transition diagrams E2 and E3 given in Figure 3.3. Table 5.5 shows the different general option trees that can be computed

for each state in the diagrams. Notice the nonterminal reference for state S2, followed by the more detailed token options for that state.

STATE	GENERAL OPTION TREES
S1	"a" "b"
S2	E3 — "f" "b"
S3	"d"
S4	"e"
S5	"c"
S6	"f" "b"
S7	"g"
S8	"c"
S9	"d"
S10	none

Table 5.5 - General Options for States of Transition Diagrams E2 and E3

Table 5.6 shows a few unacceptable input strings for the diagrams E2 and E3, along with some possible error interpretations for those strings and a comparison indicator for the preliminary Token Model.

TOKEN INPUT STRING	ERROR INTERPRETATION	NUMBER OF CHANGES MADE	CAN MESSAGE BE FOUND BY TOKEN MODEL	SAMPLE "SUGGESTION" MESSAGE
"a d"	"a E3 d"	1	no	Insert a 'E3' in front of "d."
	"a f"	1	yes	Replace "d" with "f."
	"a b"	1	yes	Replace "d" with "b."
	"a f g d"	2	yes	Insert "f" and "g" in front of "d."
"a c"	"a E3"	1	no	Replace "c" with a 'E3.'
	"a b c"	1	similar	Insert "b" in front of "c," making 'E3' be "b c."
	"b c"	1	yes	Replace "a" with "b."
"b c d"	"b c e"	1	yes	Replace "d" with "e."
	"a b c d"	2	similar	Insert "a" in front of "b," making 'E3' be "b c."

Table 5.6 - Extended Error Interpretations for Transition Diagrams E2 and E3

As another example of some of the suggestions that can be made using the extended suggestion model, consider transition diagrams E6 through E9, shown in Figure 5.9. The acceptable input strings for these diagrams are:

"a c b a e"

"a c d e"

"a d e"

"b a c e b a"

"b a c e d"

"d c e b a"

"d c e d"

The general option trees that can be computed for each state in the diagrams are shown in Table 5.7. Note that for some of the states in this table, particularly states S16 and S17, the general option tree depends on not only the transition diagram itself, but also on the parser's stack at the time when the tree is constructed. Each of these states is a final state for one of the transition diagrams, and therefore it is necessary to know from where that diagram was invoked in order to build the tree properly.

Table 5.8 shows a few unacceptable input strings for the diagrams E6 - E9, along with some possible error interpretations for those strings and some sample suggestion messages for these interpretations.

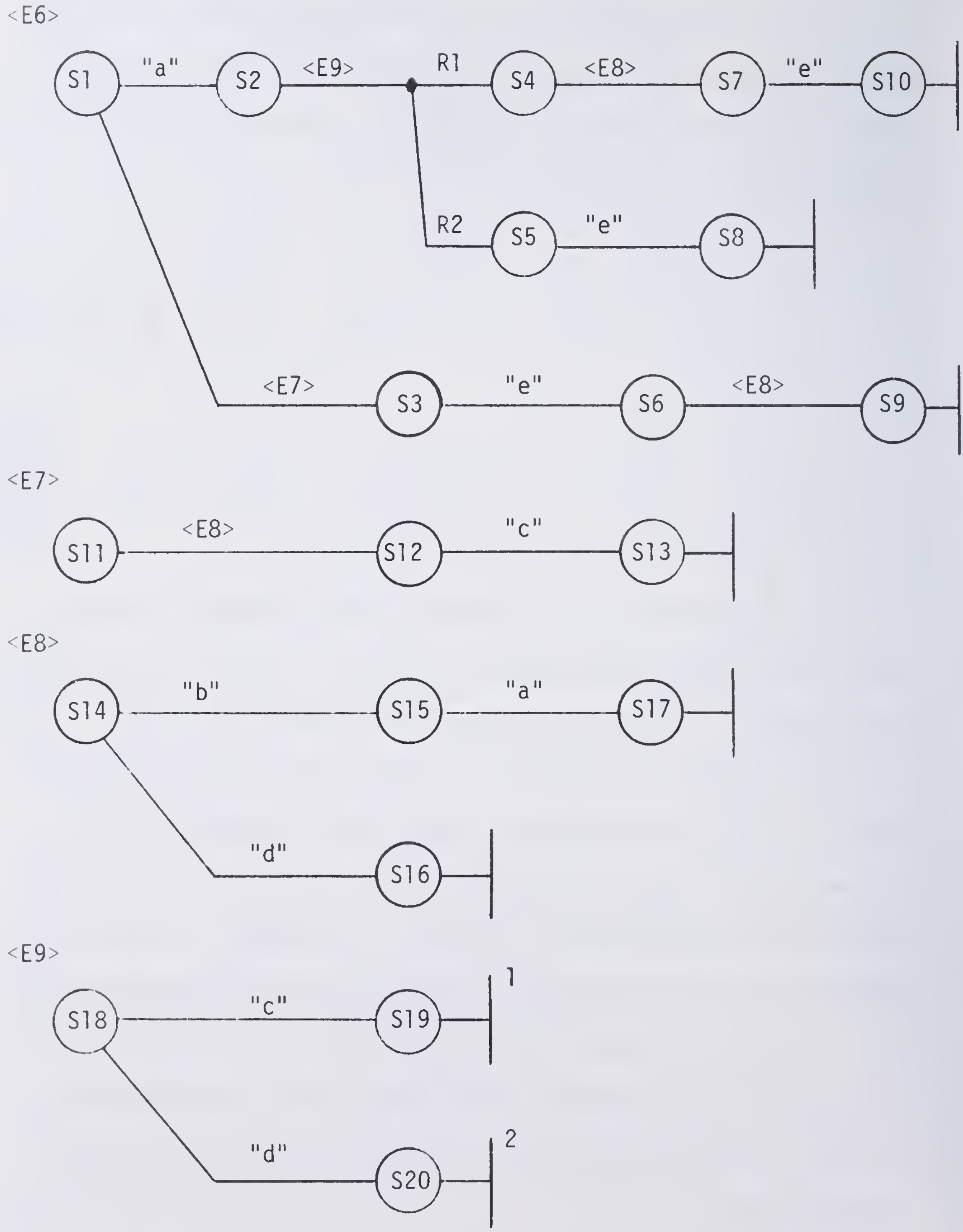


Figure 5.9 - Transition diagrams E6 - E9

STATE	GENERAL OPTION TREES	
S1	"a" <E7> - <E8> "b" "d"	
S2	<E9> "c" "d"	
S3	"e"	
S4	<E8> "b" "d"	
S5	"e"	
S6	<E8> "b" "d"	
S7	"e"	
S8, S9, S10	none	
S11	<E8> "b" "d"	
S12	"c"	
S13	"e"	: called from state S1.
S14	"b" "d"	
S15	"a"	
S16, S17	"e"	: called from state S4.
	none	: called from state S6.
	"c"	: called from state S11.

STATE	GENERAL OPTION TREES	
S18	"c" "d"	
S19	— <E8> "b" "d"	: called from state S2.
S20	— "e"	: called from state S2.

Table 5.7 - General Options for Diagrams E6 - E9

TOKEN INPUT STRING	ERROR INTERPRETATION	NUMBER OF CHANGES MADE	SAMPLE "SUGGESTION" MESSAGE
"c"	"a c"	1	Insert "a" in front of "<E9>."
	"<E7>"	1	Replace "c" with "<E7>."
	"<E8>c"	1	Insert "<E8>" in front of "c."
	"b"	1	Replace "c" with "b."
	"d c"	1	Insert "d" in front of "c."
	"b a c"	2	Insert "b" and "a" in front of "c."
"a c e"	"a c <E8> e"	1	Insert "<E8>" in front of "e."
	"a c b"	1	Replace "e" with "b."
	"a c b a e"	2	Insert "b" and "a" in front of "e."
	"a c d e"	1	Insert "d" in front of "e."
	"a d e"	1	Replace "<E9>" with "d."
	"b a c e"	1	Insert "b" in front of "a."
"b a e"	"b a c e"	1	Insert "c" in front of "e."
	"a <E9> b a e"	2	Insert "a" and "<E9>" in front of "b."

Table 5.8 - Some Extended Error Interpretations for Diagrams E6 - E9

5.4 Overall Control of the Extended Model

One final problem concerning the overall control and operation of the error system has yet to be completely discussed. The MODIFY routine, which essentially controls the entire error analysis process, is always defined in terms of some starting input token location and corresponding parser state. The REPARSE routine, if it recursively invokes the MODIFY routine, does so with the location and state parameters updated for the location of the next modification that must be attempted. However, the overall error system's initial invoking of MODIFY, although briefly mentioned in section 5.2.2, is as yet unspecified.

The algorithm that is proposed is to make the initial activation of the MODIFY routine at the actual original error token location and the state of the parser where the original error was detected. This corresponds to the following:

//MODIFY (error_token_location, corresponding_state, 0)//.

This initial call assumes that the error token itself is in error, and the MODIFY routine's analysis will generate suggestions about exactly what the parser was expecting when the error was detected. Since all modifications will be tried at the very end of the input string, <u>any</u> <u>R</u>eplace operation that is tried will yield a "corrected" interpretation immediately. It is useful to

have the first "possible correction" suggestions provide information about the state of the parse when the error was detected.

After all possibilities have been tried from this initial location, the MODIFY routine will return to the error_-system_controller. Then the controller should reinvoke the MODIFY routine from a location and corresponding parser state that is one token back in the input string. Likewise, each time that the MODIFY routine exhausts all possibilities from one location, the controller should move back another token and reactivate the MODIFY routine. Each of these reactivations makes the assumption that the original error token is itself "correct," and that the actual syntax error occurred someplace back in the input string. All of the modifications are made to things located before the original error token, which is not disturbed, but is left intact at the right end of the input string.

As mentioned in section 5.2.2, the backup process should terminate when the left context "beacon" is reached.

There is one more important parameter that must be controlled by the error analysis system: that is the upper limit of modifications, N, that can be made while constructing any error interpretation. This limit is checked in the REPARSE routine when the decision about allowing another modification to be made must be resolved. There are two approaches that can be taken to control N. One is to compute all suggestion messages from one location with

1, 2, . . ., N modifications allowed (for some predetermined N), and then to move back a token and repeat the process. The other approach is to compute all error interpretations of the input string with one modification, then compute all interpretations with two modifications, etc.

This thesis proposes that the second approach be utilized for the following reason. In the interactive compiling environment described in Chapters 1 and 3, it was noted that as the programmer enters the program, it is syntactically parsed, and errors are signalled as soon as detected. This implies that it is likely that the actual input string is fairly "close" to being correct when an error is detected. Therefore, it seems better to compute the "close" error interpretations first (i.e., the N limit equalling one, then two, etc.), before attempting to find an interpretation with a larger number of required modifications.

Figure 5.10 gives the ERROR_SYSTEM_CONTROL routine. This routine is the error system module that is invoked by the parser system (SYNA) when the original error is detected. It controls the number N of modifications that are allowed in any reparse attempt and also where each reparse attempt originates. Then each reparse attempt that is originated is successful only if the input string can be corrected with exactly N modifications; the MOD_TREE, MODIFY, REPARSE, and USER_INTERACTION routines are still as given in Figures 5.8, 5.7, 5.6, and 5.5, respectively.

```
-   begin  ERROR_SYSTEM_CONTROL
    comment  This routine is invoked by SYNA when a parser syntax
             error is originally detected.  The routine provides
             control over the initiation of reparse attempts, and
             the number of modifications allowed in any interpre-
             tation.
-   /maximum_N ← predetermined maximum number of allowed changes
               in any interpretation/
-   /establish the error environment/
-   /beacon_location ← farthest token in the input that will be
                       backed-up to/
-   loop for  N = 1 step  1 to maximum_N :
        comment  set up initial invoking locations.
-       loc ← error_token_location
-       state ← parser state associated with error_token_location
-       loop :
            comment  Initiate reparse attempt.
-           //MODIFY (loc, state, 0)//
-       while  loc > beacon_location :
            comment  Initiate reparse attempt from new location
-           /back up 1 token in the input string/
-           /loc ← new input location/
-           /state ← corresponding state/
-       repeat
```

```
-     repeat
      comment  No more help is available.
-     /Report to user that no more help is available/
-     /reestablish the compiler's editing environment/
-     end  ERROR_SYSTEM_CONTROL
```

Figure 5.10 - ERROR_SYSTEM_CONTROL routine

Chapter 6

CONCLUSIONS AND FUTURE RESEARCH

6.1 Summary

This thesis describes an interactive compiler diagnostic system. It is assumed that the compiler will be used by elementary programmers, and the responsibility of the diagnostic system is to behave like a "consultant" to these programmers with respect to syntactic errors. It is important that this consulting system be very easy to use and interact with so as to avoid confusing or frustrating the programmer. In addition, it is important that any diagnostic messages that are given to the programmer be very clearly stated using terminology with which the programmer is familiar.

The diagnostic consulting system communicates with a programmer about an error by suggesting "possible corrections" that can be made to the program to yield a valid syntactic prefix in the programming language. These suggestions are determined by examining the program input string, the parser's internal state information, and the parser's table; this implies a programming-language-independent diagnostic system. The only information required by the diagnostic system in addition to this normal compiler information is an "English phrase" table which provides readable

English character strings that can be used to describe various pieces of parser information to the programmer.

The system operates by successively moving farther and farther to the left of the position in the input string where the original error was detected; at each successive position, a modification to the input string is made (replacement, addition, or suppression) and the remainder of the input up to and including the original error token is reparsed. If no new errors are detected during this reparsing, then the modified input string is syntactically correct and a "possible correction" suggestion message can be given to the programmer that simply describes what modifications were made. If a new error is detected during the reparsing, then additional or different modifications must be made to the input string and the reparsing attempted again.

Unlike previous systems, the modifications made to the input string are concerned not only with simple tokens, but also with any nonterminals that are acceptable at that position in the parse. These nonterminals generally represent higher-level constructs in the programming language, and thus suggestion messages can be given using this higher-level terminology. A further advantage of these higher-level suggestions is that successive suggestions will frequently provide more and more detail about the particular nonterminal until the detail reaches the actual "token" level; thus, the suggestions tend to at first provide higher-level

information, followed by successively more refined and detailed information for the programmer. It is very easy, in addition, to provide the programmer with some control over the amount of detail that is presented; the programmer can, at any time, request that the successively refined suggestions not be presented, but that other, different suggestions be given instead (if possible).

6.2 Evaluation

6.2.1 Implementation Performance

As mentioned in Chapter 1, one of the key objectives of this research was to create the first interactive diagnostic consulting system by actually implementing on the PLATO IV system the algorithms that have been described. The entire diagnostic system was written in TUTOR [She., '74], a Fortran-level language supported by PLATO. TUTOR is a very good language for input/output: it contains very powerful facilities for accepting and examining keyboard input from the interactive terminal and for preparing arbitrarily complex graphics displays on the terminal's screen. However, it is fairly primitive with respect to program control constructs and memory management; in particular, it does not support dynamic storage allocation, and therefore recursion is not available. Also the subroutine return-address stack is only ten levels high, a limit that is easy to exceed in a complex TUTOR

program. The implementation was completed, however, in approximately 6 months, of which about 3 months were full-time (summer) and the remainder part-time.

Because of the nonrecursive limitation of TUTOR, the actual prototype system was restricted somewhat from the algorithms described in Chapter 5. The most important restriction is that N, the number of modifications allowed in any interpretation, is limited to the value of N = 1. This means that after the first modification has been made in a possible interpretation, the remainder of the input string must be acceptable without any changes for the reparse attempt to succeed. This removes the recursion requirement for the MODIFY routine immediately. It also allows the REPARSE routine to be written without recursion, since then its only responsibility is to simply check whether any new errors are signalled if the parser is started on the remainder of the input string. If no error is signalled, the reparsing succeeds; otherwise the single modification that was made did not correct the string and REPARSE must return to MODIFY so that a different modification can be tried.

The only disappointment has been the fairly slow response time of the diagnostic system when another suggestion is requested. Since determining suggestions requires examining the parser's table and repeatedly reparsing the modified input string, the diagnostic system is generally compute-bound. Unfortunately, the PLATO IV

system is designed for input/output bound programs with time for relatively few calculations.

The response time problem is not limited to the diagnostic system, however; the entire compiler system exceeds the currently suggested cpu usage limits of 2 - 4 ms/sec. The most recent timing estimates indicate that the overall compiler runs very well at 7 - 8 ms/sec., and marginally well at 5 - 6 ms/sec. In comparison, the diagnostic system runs very well at 10 - 12 ms/sec., and acceptably at about 8 - 10 ms/sec.; the performance rapidly deteriorates below these levels, however, and subjective tests indicate that the diagnostic system is too frustrating and slow to be useful with a cpu limit below 5 - 6 ms/sec.

An important observation about the operation of the diagnostic system is that the initial suggestion messages that are given by the system are generally quite easy to determine and compute since very little context is examined (i.e., only the actual token that originally caused the error to be signalled). Therefore, the first few "possible correction" messages can be computed for the programmer with a small cpu limit (around 4 - 5 ms/sec.), so that at least some information is available even when the PLATO IV system is fairly heavily loaded. It is mostly when suggestions concerning modifications farther back in the input string must be computed and PLATO IV is heavily loaded that intolerable response delays may occur.

6.2.2 Effectiveness

At the present time, because of the relatively slow response times mentioned above, the diagnostic system has not been used by elementary programmers; therefore it is impossible to report on the actual demonstrated effectiveness of the system. It is possible, however, to compare a few of the general characteristics of the consulting system with other diagnostic systems.

The diagnostic systems for the PLATO IV compilers were originally designed as a set of *ad hoc*, hand-coded, one-line error messages, determined by a specific error number that was obtained from the parser's table. One specific error message was displayed upon detection of an error in the user's program, and this message usually consisted of describing the most likely token (in the language implementor's opinion) that would be acceptable at that point in the parse. The diagnostic system proposed in this thesis is language independent (it does not use any special error numbers, but determines messages by examining the productions in the parser's table), and the first few messages to the programmer always mention the tokens or nonterminals that could be used immediately at the location of the error in the program. These initial messages therefore will usually provide not only the same information that was available using the hand-coded technique, but actually all of the correction possibilities. Since these first few messages are

easy to compute, as mentioned previously, the consulting diagnostic system seems to provide at least as effective (frequently more effective) assistance as did the *ad hoc* systems.

To determine how many immediate correction possibilities there are for a particular error situation, it is necessary to examine the general option tree for the state in which the parser detected the error. A study of the general option trees for various parser states was made to discover some of the characteristics of these trees. The small subset of PL/I that has been implemented (assignment, DECLARE, GOTO, IF, DO, list I/O, CALL, and RETURN statements) was used for the study. The detailed results are given in Appendix I. It was found that about 65 percent of the trees for the possible parser states contained only one or two option nodes. About 25 percent of the trees contained nonterminal option nodes, and the maximum height of the trees was four (indicating up to three nonterminal option nodes in a tree). Roughly 65 percent of the trees had only one or two token option nodes; however, there were also a number of states with eight to ten token options (primarily states accepting numeric expressions) and eighteen to twenty token options (for the beginning of a new statement).

In interpreting this information, it is important to remember that the statistics are a measure of the trees for most of the possible parser states that can occur in the PL/I subset.

However, in any actual PL/I program, some of the possible states occur much more frequently than others. In particular, states corresponding to the start of a new statement or states accepting numeric expressions occur frequently. Therefore, the trees for these frequently-encountered states will be used most often. Since the trees for statements and expressions are among the largest and highest for PL/I, the consulting system also has the most information available for these states.

If the consulting system is compared to the system of Levy [Lev, '71], it is interesting to note that near the original error token the consulting system can provide outstanding suggestions. Not only are simple "token" modifications suggested (as is done by Levy), but also appropriate nonterminal modifications are given. Furthermore, these nonterminal messages are then successively refined down to the actual token level, with the programmer having some control over this refinement process. These suggestions seem to provide much better assistance to a programmer than just the simple token messages provided by Levy.

As suggestions are determined about modifications farther back in the input string, however, the two systems appear to produce quite similar suggestions. These suggestions concern only simple <u>token</u> modifications; the consulting system is seldom able to make suggestions involving <u>nonterminals</u> farther back in the input. This can be explained by realizing that acceptable strings in a

programming language are usually syntactically "far apart" or dissimilar. If a nonterminal is recognized in the parsing of a string, there is a high probability that that nonterminal was intended by the programmer and that it is formed and located correctly. Therefore, if an error is encountered to the right of a recognized nonterminal, it will seldom be possible to modify that <u>nonterminal</u> and thereby obtain a legal interpretation. However, modifying a single token <u>before</u> the nonterminal may cause the tokens comprising the original nonterminal to be reparsed differently, yielding a legal interpretation.

6.3 Future Research

There are a number of areas that can be suggested for future research. One project is to continue to work on and optimize the TUTOR-coded implementation on PLATO IV. The implemented system would be much more useful if it were a factor of two or three times more efficient. Work is already underway by the PLATO systems staff to modify some of the TUTOR commands to allow a much more efficient implementation. There is no doubt, also, that the current program code could be improved substantially.

Another area of research that definitely could be approached is to actually test the usefulness and effectiveness of the diagnostic system with programmers. The system could be statistically compared with other existing systems to determine the

relative effectiveness of the systems for elementary programmers. It would also be very interesting to determine the usefulness of the consulting system for advanced programmers.

A final research suggestion is to look at the problem of explaining the effect of a "possible correction" suggestion on the program. The proposed system is very good at describing different ways to correct the program, but it relies on the programmer to examine the suggestions and determine what the ramifications of making a particular correction will be on the program. Further work on the characterization and description of context-sensitive semantic requirements would also be very useful in attempting to explain the effect of a particular correction on a program.

6.4 Conclusion

The use of interactive computing systems is steadily increasing in the computing world. It is anticipated that interactive, incremental compilers will become standard features of future systems. This thesis has described one possible approach to the handling of diagnostic assistance in this environment. It is hoped that this work will help stimulate research in this area.

LIST OF REFERENCES

[Aho., '72] : Aho, A. V. and Ullman, J. D., The Theory of Parsing Translation, and Compiling, Volume 1, Prentice-Hall, Inc., 542 pp., 1972.

[Alp., '70] : Alpert, D. and Bitzer, D., "Advances in Computer Based Education," Science, Vol. 167 (20 March, 1970), pp. 1582-1590.

[Con., '63] : Conway, M., "Design of a Separable Transition-Diagram Compiler," CACM, Vol. 6 (July 1963), pp. 396-408.

[Con., '73] : Conway, M., and Wilcox, T., "Design and Implementation of a Diagnostic Compiler for PL/I," CACM, Vol. 16, No. 3 (March 1973), pp. 169-179.

[Cre., '70] : Cress, P., Dirksen, P., and Graham, J., Fortran IV with Watfor and Watfiv, Prentice-Hall, Inc., 1970.

[Dav., '75] : Davis, A., An Interactive Analysis System For Execution-time Errors, Ph.D thesis, Department of Computer Science, University of Illinois, January 1975, Report # UIUCDCS-R-75-695.

[DeR., '71] : DeRemer, F. L., "Simple LR(k) Grammars," CACM, Vol. 14, No. 7 (July 1971), pp. 453 - 460.

[Ela., '75] : Eland, D., Forthcoming Ph.D thesis on the GUIDE information-retrieval system, to be published summer 1975.

[Flo., '63] : Floyd, R. W., "Syntactic Analysis and Operator Precedence," JACM 10 (July 1963), pp. 316 - 333.

[Gra., '73] : Graham, S. and Rhodes, S., "Practical Syntactic Error Recovery in Compilers," Conference Record of the ACM Symposium on the Principles of Programming Languages, Boston, Mass., October 1973, pp. 52-58.

[Gri., '74] : Griffiths, M., "LL(1) Grammars and Analyzers," Compiler Construction, Bauer, F. and Eickel, J. (eds), Springer-Verlag, New York, 1974, pp. 57-83.

[Iro., '63] : Irons, E., "An Error Correcting Parse Algorithm," CACM, Vol. 6 (Nov. 1963) pp. 669-673.

[JaE., '73] : James, E., and Partridge, D., "Adaptive Correction of Program Statements," CACM, Vol. 16, No. 1 (Jan. 1973), pp. 27-37.

[JaL., '72] : James, L., A Syntax-directed Error Recovery Method, Masters thesis, Computer Systems Research Group, University of Toronto, May 1972, # CSRG-13.

[LaF., '70] : LaFrance, J., "Optimization of Error Recovery in Syntax-directed Parsing Algorithms," Sigplan Notices 5 (Dec. 1970) pp. 2-17.

[LaF., '71] : LaFrance, J., Syntax-directed Error Recovery for Compilers, Ph.D thesis, Department of Computer Science, University of Illinois, 1971, # 459.

[Lei., '70] : Leinius, R., Error Detection and Recovery for Syntax Directed Compiler Systems, Ph.D thesis, Computer Science Department, University of Wisconsin, 1970.

[Lev., '71] : Levy, J., Automatic Correction of Syntax Errors in Programming Languages, Ph.D thesis, Computer Science Department, Cornell University, Dec. 1971, Technical Report # TR 71-116.

[Lom., '73] : Lomet, D., "A Formalization of Transition-diagram Systems," JACM, Vol. 20, No. 2, (April 1973), pp. 235-257.

[Lyo., '74] : Lyon, G., "Syntax-directed Least-errors Analysis for Context-free Languages: A practical Approach," CACM, Vol. 17, No. 1, (January 1974), pp. 3-14.

[Nie., '74] : Nievergelt, J., Reingold, E., and Wilcox, T., "The Automation of Introductory Computer Science Courses," A. Gunther, et al. (eds), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

[Rho., '73] : Rhodes, S., Practical Syntactic Error Recovery for Programming Languages, Ph.D thesis, Department of Computer Science, University of California at Berkeley, June 1973, Technical Report 15.

[She., '74] : Sherwood, B., The TUTOR Language, Computer-based Education Research Laboratory and Department of Physics, University of Illinois, Urbana, Illinois, 1974.

[Tin., '75] : Tindall, M., An Interactive Table-driven Parser System, Masters thesis, Department of Computer Science, University of Illinois, August 1975.

[Wil., '73] : Wilcox, T., "The Interactive Compiler as a Consultant in the Computer Aided Instruction of Programming," Proceedings of the Seventh Annual Princeton Conference in Information Sciences and Systems, March 1973.

[Wir., '66] : Wirth, N., and Weber, H., "EULER-A Generalization of ALGOL and its Formal Definition," CACM, Vol. 9, No. 1 (January 1966) pp. 13 - 25, and Vol. 9, No. 2 (February 1966) pp. 89 - 99.

APPENDIX I

GENERAL OPTION TREE EXAMPLES

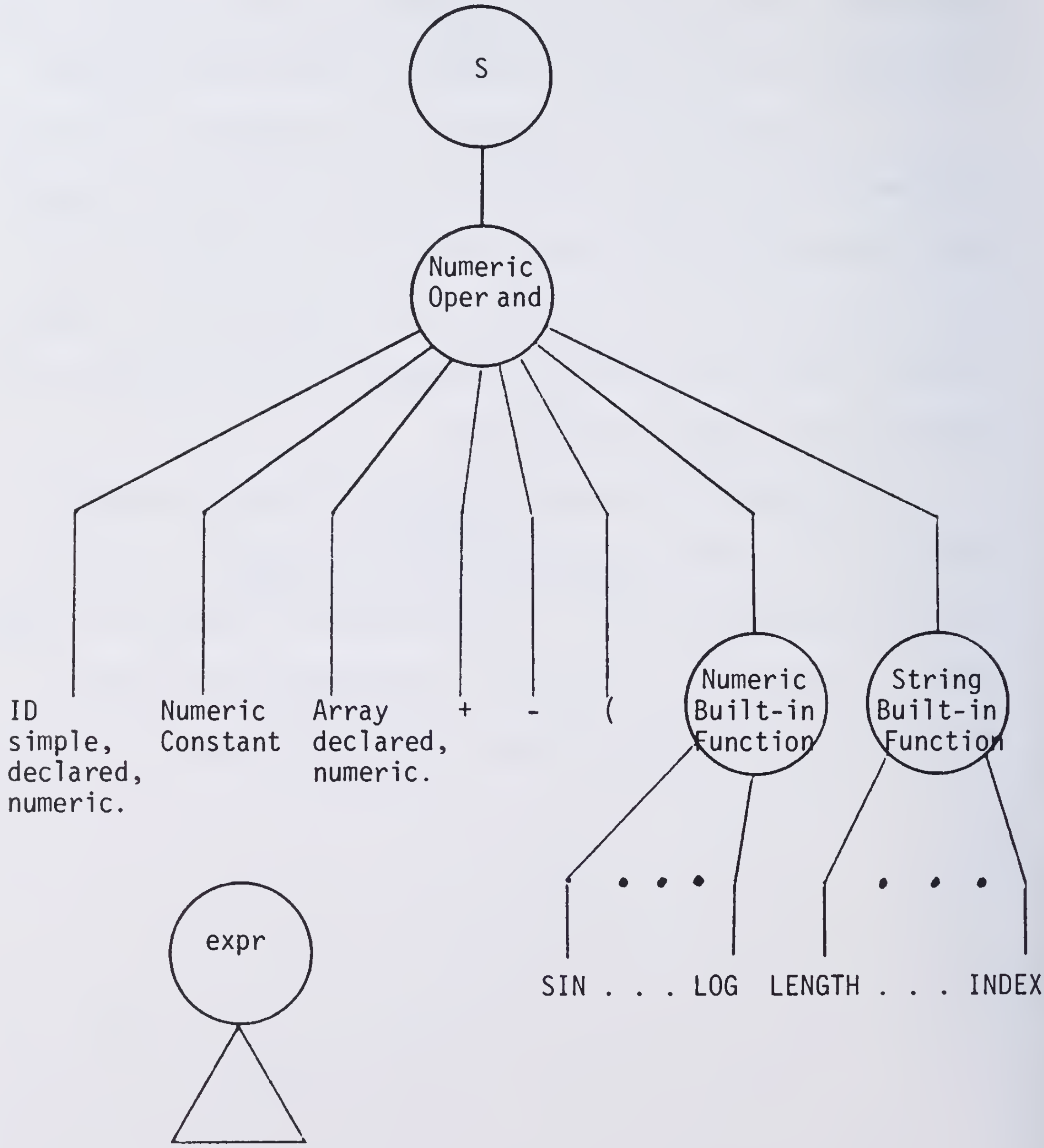
This appendix contains the results of a study of the size and shape of the general option tree for many of the possible parsing states. The analysis was performed on the small subset of the PL/I language that has been implemented on the PLATO IV system.

In each case, the information is presented as a particular input string and the corresponding option tree that is acceptable at that point in the parse of that string. Obviously, only some of the possible input strings could be examined, but a representative sample was selected and analyzed. In all, a total of 61 possible input strings were examined.

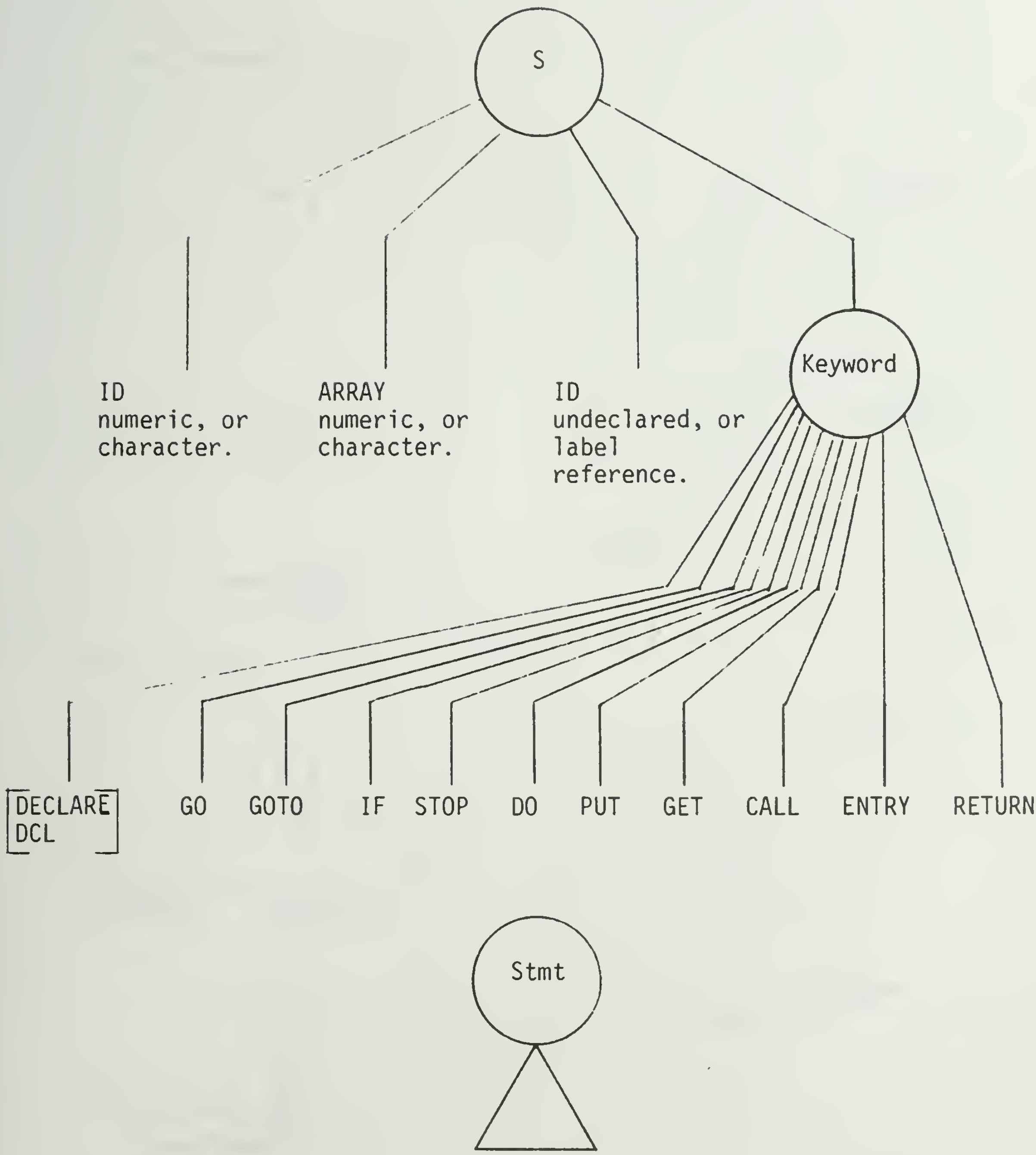
To simplify the form of the option trees, two special subtrees will be used: one for the start of a numeric expression and the other for the start of a new statement. Whenever an expression or a statement is specified as a nonterminal in an option tree, the respective special subtree will be indicated.

Special Subtrees

Numeric Expression = expr

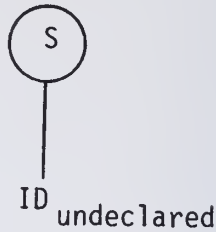


Statement = Stmt

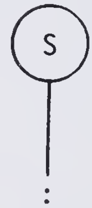


INPUT | OPTION TREE

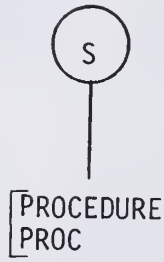
<null program>



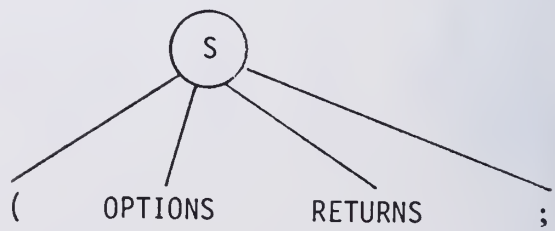
TEST10 Δ



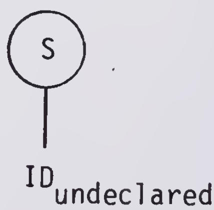
TEST10: Δ

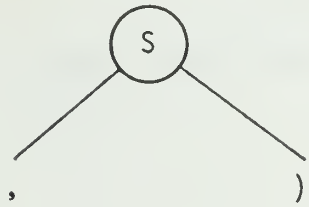
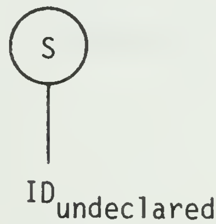
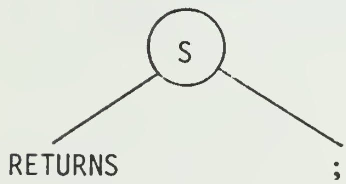
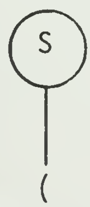


T : PROC Δ



T : PROC (Δ



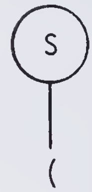
INPUT	OPTION TREE
T : PROC (A$_{\Delta}$	
T : PROC (A,$_{\Delta}$	
T : PROC (A)$_{\Delta}$	
T : PROC OPTIONS$_{\Delta}$	
T : PROC OPTIONS ($_{\Delta}$	
T : PROC OPTIONS (MAIN$_{\Delta}$	

INPUT

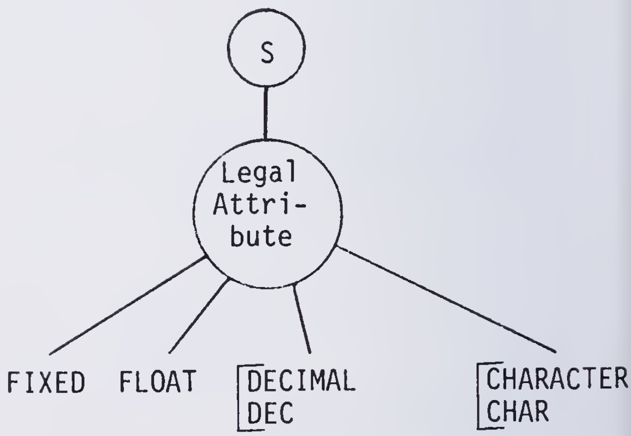
OPTION TREE

T : PROC OPTIONS (MAIN)$_{\Delta}$

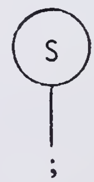
T : PROC RETURNS$_{\Delta}$



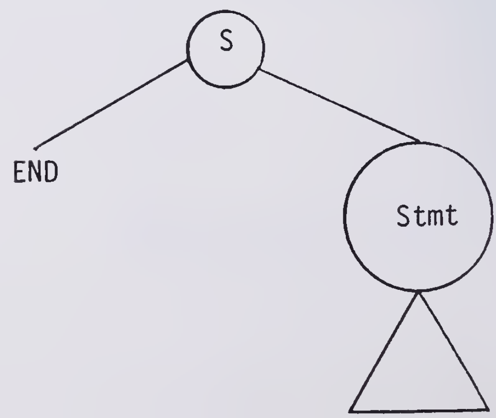
T : PROC RETURNS ($_{\Delta}$



T : PROC RETURNS (CHAR)$_{\Delta}$



T : PROC ;$_{\Delta}$

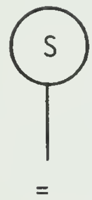


INPUT

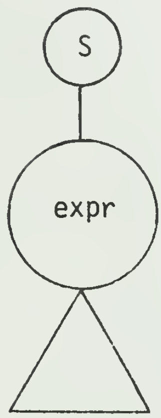
OPTION TREE

ASSIGNMENT STATEMENTS

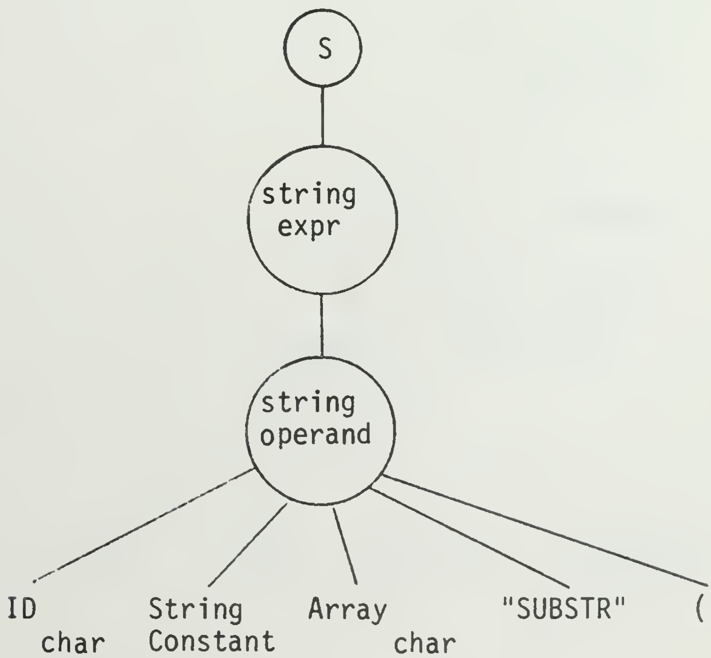
ID
numeric, or Δ
character



ID =
numeric Δ



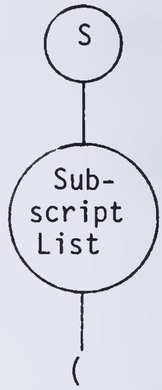
ID =
character Δ



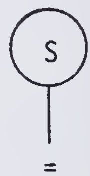
INPUT

OPTION TREE

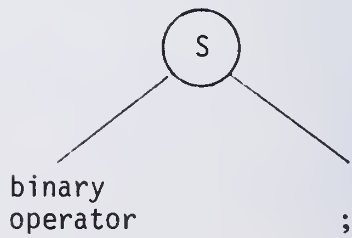
DCL A(10);
A_Δ



DCL A(10);
A $(5)_\Delta$

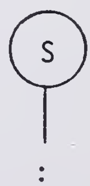


$ID_{numeric} = 10_\Delta$



LABELS

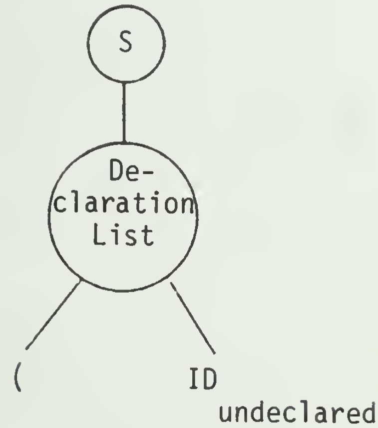
ID
  label. ref, or Δ
  undeclared



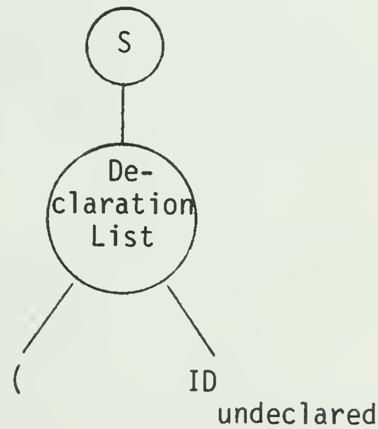
<u>INPUT</u> <u>OPTION TREE</u>

<u>DECLARE STATEMENTS</u>

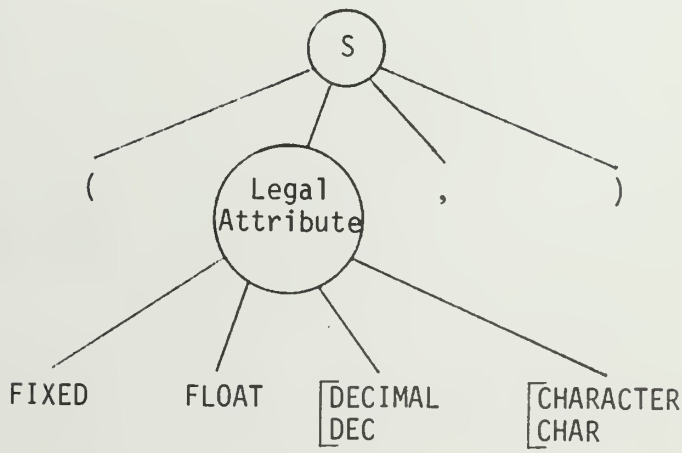
DCL$_\Delta$



DCL ($_\Delta$

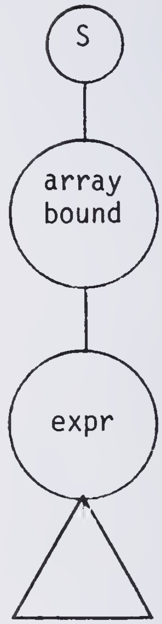


DCL (A$_\Delta$

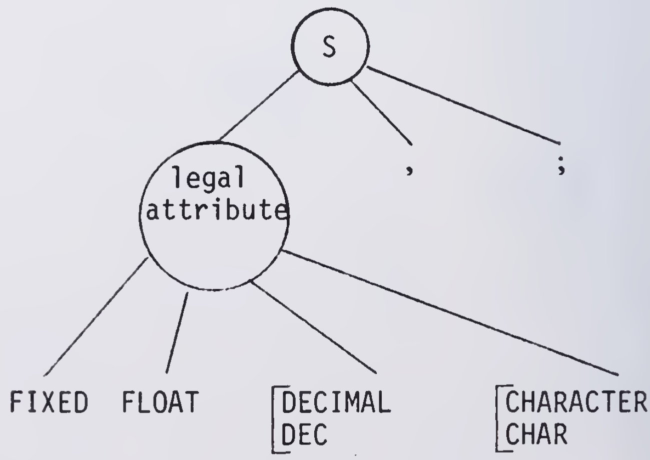


INPUT | OPTION TREE

DCL (A($_\Delta$



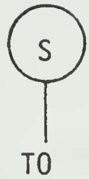
DCL (A)$_\Delta$



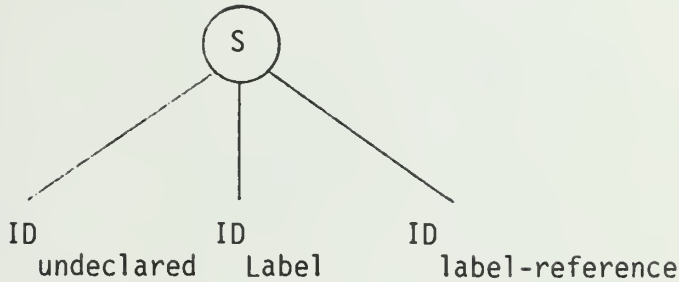
INPUT | OPTION TREE

GO Statements

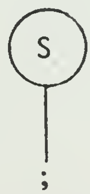
GO $_{\Delta}$



GOTO $_{\Delta}$



GOTO L1$_{\Delta}$

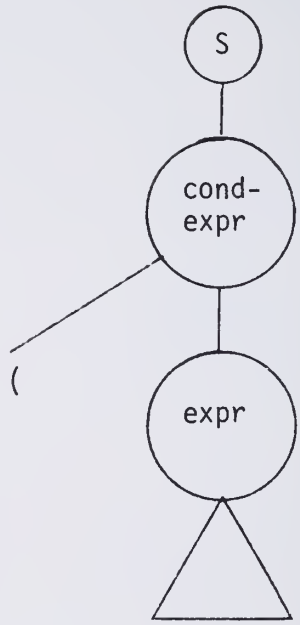


INPUT

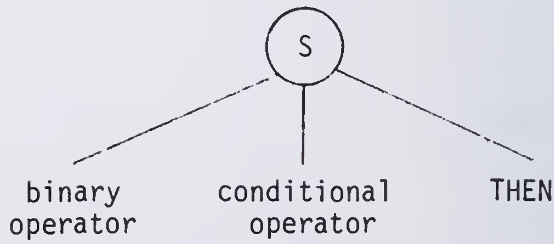
OPTION TREE

IF Statements

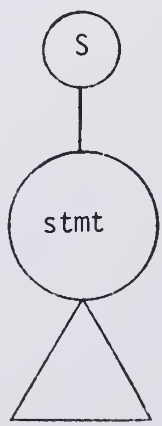
IF_{Δ}



IF I = 1_{Δ}

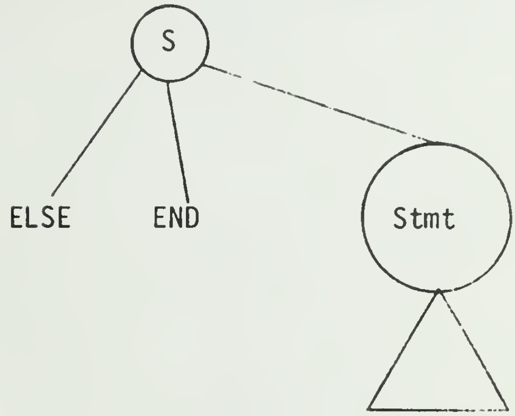


$\text{IF I = 1 THEN}_{\Delta}$



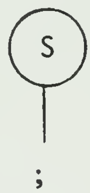
INPUT | OPTION TREE

IF I = 1 THEN STOP; Δ



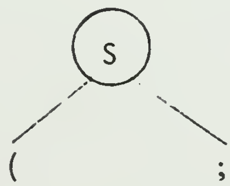
STOP Statement

STOP Δ

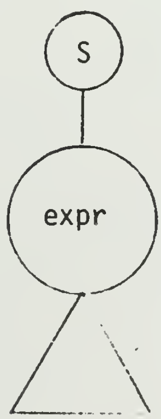


RETURN Statement

RETURN Δ



RETURN (Δ

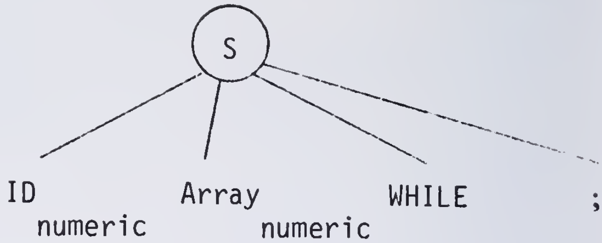


INPUT

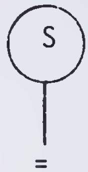
OPTION TREE

DO Statement

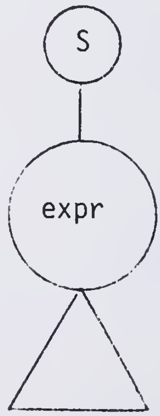
DO$_{\Delta}$



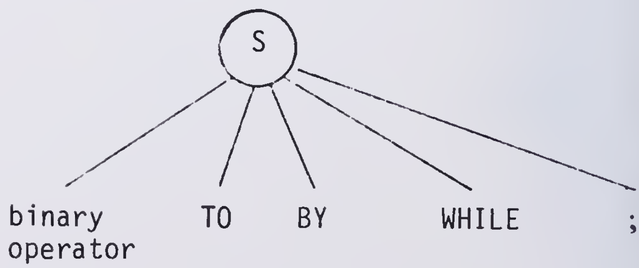
DO I$_{\Delta}$



DO I = $_{\Delta}$

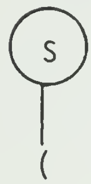


DO I = 1$_{\Delta}$

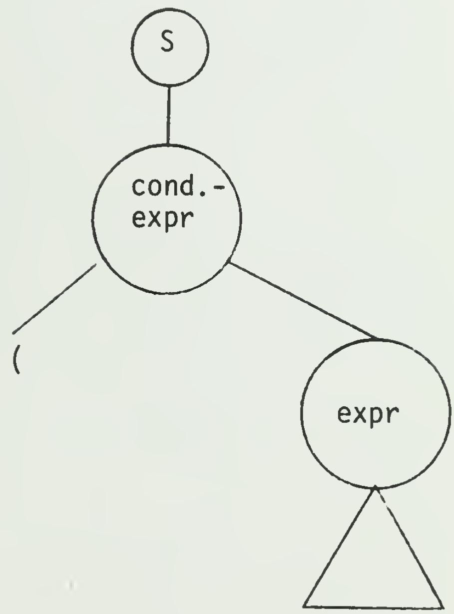


<u>INPUT</u> <u>OPTION TREE</u>

DO WHILE$_{\Delta}$

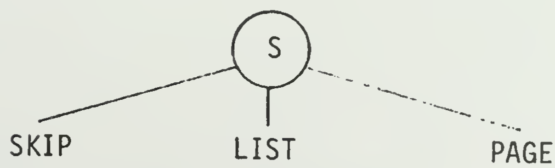


DO WHILE ($_{\Delta}$

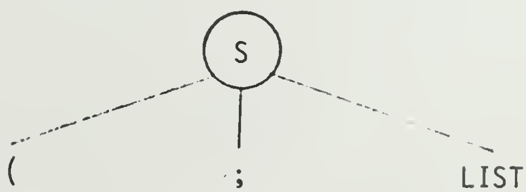


<u>PUT Statement</u>

PUT$_{\Delta}$

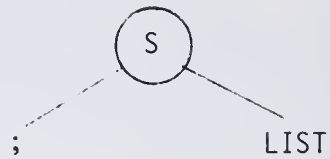


PUT SKIP$_{\Delta}$



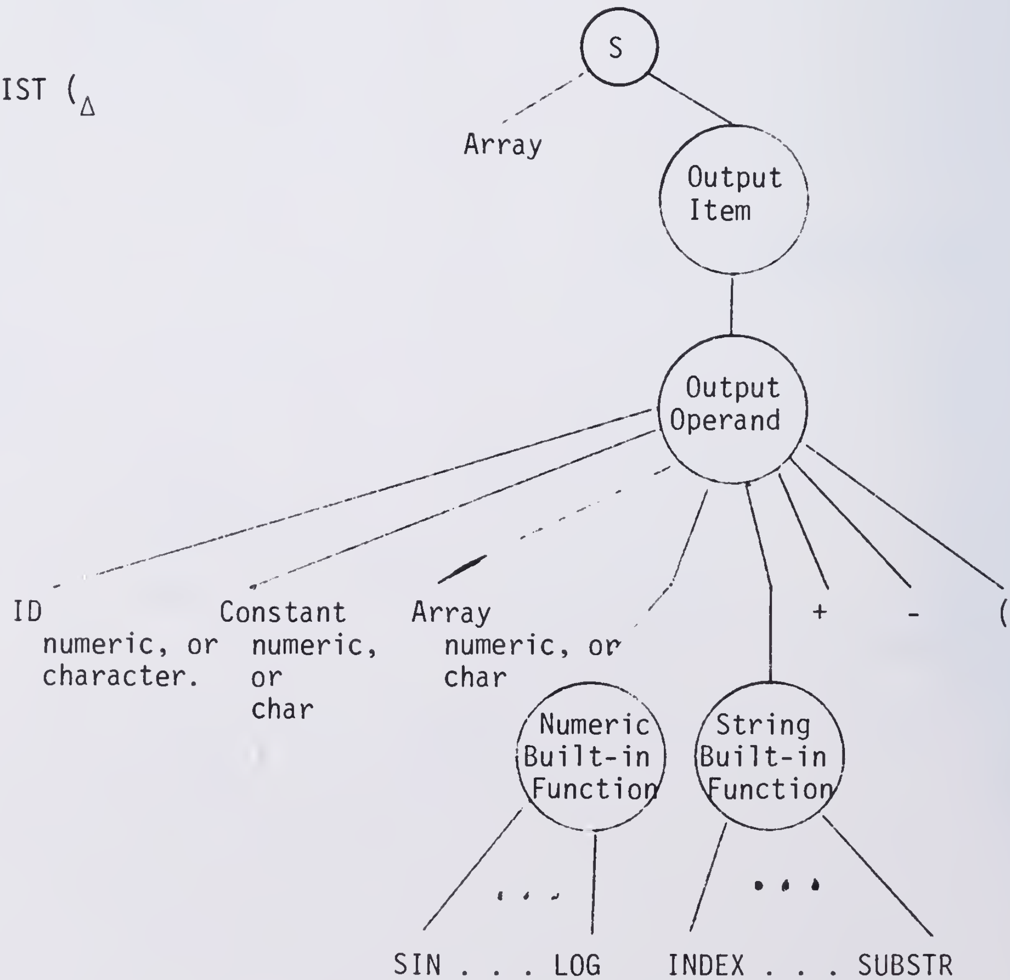
INPUT | OPTION TREE

PUT PAGE$_\Delta$



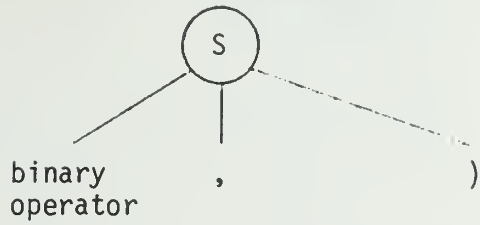
PUT LIST$_\Delta$

PUT LIST ($_\Delta$

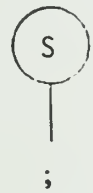


<u>INPUT</u> <u>OPTION TREE</u>

PUT LIST $(I_{\Delta}$

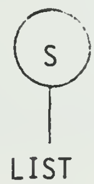


PUT LIST $(I)_{\Delta}$

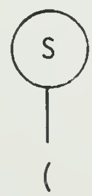


<u>GET Statement</u>

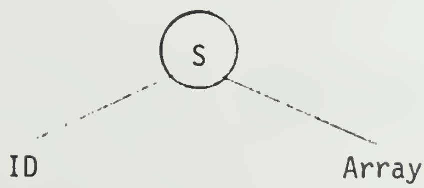
GET $_{\Delta}$



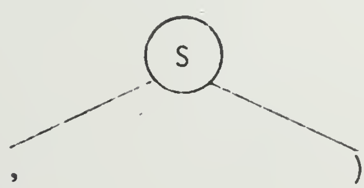
GET LIST $_{\Delta}$



GET LIST $(_{\Delta}$



GET LIST $(I_{\Delta}$

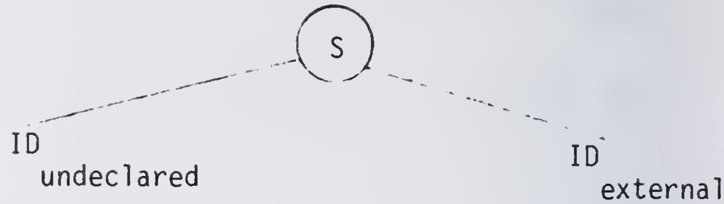


INPUT

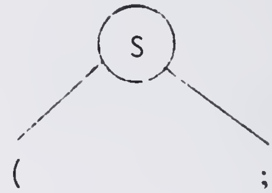
OPTION TREE

CALL Statement

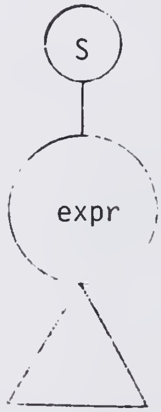
CALL $_\Delta$



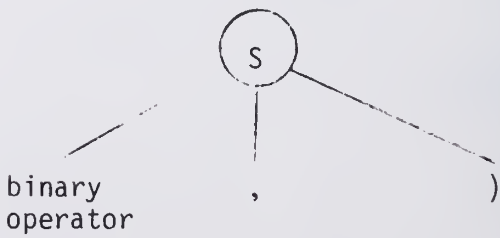
CALL B$_\Delta$



CALL B ($_\Delta$



CALL B (I$_\Delta$



CALL B (I)$_\Delta$

APPENDIX II

EXAMPLES OF "POSSIBLE CORRECTION" SUGGESTIONS FOR PL/I

This appendix contains a number of examples of the "possible correction" suggestions that are generated by the diagnostic system that has been implemented on PLATO IV. All of the examples are taken from the subset of the PL/I language that is in use by the interactive computer.

Each example is presented as a number of consecutive diagrams or "frames." The person using the diagnostic system would actually see only one frame at a time; each successive frame would be displayed as the user pressed the appropriate "HELP" or "Shifted-HELP" Key. The different frames for a particular error situation are presented on the same page in this appendix only as a convenience to the reader.

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 284

.A: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED . . .[]

**********ERROR**********                     (BACK) or (ERASE) to fix.
 This undeclared variable is not permitted here.
   Press (HELP) for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 284

.A: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED . . .[]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [    ] with an attribute.
   Press (HELP) for a different suggestion.
   Press (SHIFT)(HELP) to see a legal attribute.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 284

.A: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED . . .[]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [    ] with an attribute "DECIMAL".
   Press (HELP) for a different suggestion.
   Press (SHIFT)(HELP) to see another legal attribute.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 284

A: ......PROC;
.........DCL I FIXED...

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [    ] with an attribute "DEC".
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 284

A: ......PROC;
.........DCL I FIXED...

********** Possible Correction ********** BACK or ERASE to fix.
 Insert "," in front of [    ].
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 284

A: ......PROC;
.........DCL I FIXED...

********** Possible Correction ********** BACK or ERASE to fix.
 Insert ";" in front of [    ].
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 284
.A:......PROC;
.........DCL.I.[FIXED]...C

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with ",".
   Press (HELP) for a different suggestion.
```

```
FILE = workspace            PL/I WORKSPACE (14-new)        SPACE = 283

B: . . . . . . .PROC;
. . . . . . . . .DCL .I, . .A( . .)

**********ERROR**********                        BACK or ERASE to fix.
 This punctuation symbol is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace            PL/I WORKSPACE (14-new)        SPACE = 283

B: . . . . . . .PROC;
. . . . . . . . .DCL .I, . .A( . .)

********** Possible Correction ********** BACK or ERASE to fix.
 Insert an array bound in front of [  ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine an array bound.
```

```
FILE = workspace            PL/I WORKSPACE (14-new)        SPACE = 283

B: . . . . . . .PROC;
. . . . . . . . .DCL .I, . .A( . .)

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a numeric expression in front of [  ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine a numeric expression.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a numeric operand in front of [   ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine a numeric operand.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a declared variable ( numeric) in front of [   ].
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a numeric constant in front of [   ].
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a declered array ( numeric).
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 283

B:.......PROC;
.........DCL.I,..A(..]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an arithmetic operator.
   Press HELP for a different suggestion.
   Press SHIFT HELP to see a legal arithmetic operator.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 283

B:.......PROC;
.........DCL.I,..A(..]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an arithmetic operator "+".
   Press HELP for a different suggestion.
   Press SHIFT HELP to see another legal arithmetic operator.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 283

B:......PROC;
.........DCL.I,..A(..]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an arithmetic operator "-".
   Press HELP for a different suggestion.
```

```
FILE = workspace           PL/I WORKSPACE (14-new)         SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with "(".
   Press HELP for a different suggestion.
```

```
FILE = workspace           PL/I WORKSPACE (14-new)         SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a numeric built-in function.
   Press HELP for a different suggestion.
   Press SHIFT HELP to see a legal numeric built-in function.
```

```
FILE = workspace           PL/I WORKSPACE (14-new)         SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a numeric built-in function "ABS".
   Press HELP for a different suggestion.
   Press SHIFT HELP to see another legal numeric/built-in function.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a string built-in function.
   Press HELP for a different suggestion.
   Press SHIFT HELP to see a legal string built-in function.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 283

B:.......PROC;
.........DCL.I,..A(..[]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a string built-in function "LENGTH".
   Press HELP for a different suggestion.
   Press SHIFT HELP to see another legal string built-in function.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C:.......PROC;
.........DCL.I,.A(1Ø).DEC;
.........GET.LIST(.A.);
.........I.=.A.[;]

**********ERROR**********                       BACK or ERASE to fix.
 This punctuation symbol is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C:.......PROC;
.........DCL.I,.A(1Ø).DEC;
.........GET.LIST(.A.);
.........I.=.A.[;]

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a subscript list in front of [  ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine a subscript list.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C:.......PROC;
.........DCL.I,.A(1Ø).DEC;
.........GET.LIST(.A.);
.........I.=.A.[;]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [  ] with "(".
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C: . . . . . . .PROC;
. . . . . . . . .DCL .I, .A(10) .DEC;
. . . . . . . . .GET .LIST(.A.);
. . . . . . . . .I .= .A.;

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a numeric expression.
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C: . . . . . . .PROC;
. . . . . . . . .DCL .I, .A(10) .DEC;
. . . . . . . . .GET .LIST(.A.);
. . . . . . . . .I .= .A.;

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a numeric operand.
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C: . . . . . . .PROC;
. . . . . . . . .DCL .I, .A(10) .DEC;
. . . . . . . . .GET .LIST(.A.);
. . . . . . . . .I .= .A.;

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a declared variable ( numeric).
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 262

C:.......PROC;
.........DCL.I,.A(1Ø).DEC;
.........GET.LIST(.A.);
.........I.=.A.;

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [    ] with a numeric constant.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 264

.D: ......PROC;
.........DCL .I, .A(10);
.........GET .LIST . (A);
.........I= .A( .[)]

**********ERROR**********                     BACK or ERASE to fix.
 This punctuation symbol is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 264

.D: ......PROC;
.........DCL .I, .A(10);
.........GET .LIST . (A);
.........I= .A( .[)]

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a numeric expression in front of [  ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine a numeric expression.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 264

.D:......PROC;
.........DCL.I,.A(10);
.........GET.LIST.(A);
.........I=.A(.

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a numeric operand in front of [   ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine a numeric operand.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 262

.E:......PROC;
.........DCL.I,.A(1Ø);
.........GET.LIST.(A);
.........I=.A(.1..[;]

**********ERROR**********                    BACK or ERASE to fix.
 This punctuation symbol is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 262

.E:......PROC;
.........DCL.I,.A(1Ø);
.........GET.LIST.(A);
.........I=.A(.1..[;]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an arithmetic operator.
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 262

.E:......PROC;
.........DCL.I,.A(10);
.........GET.LIST.(A);
.........I=.A(.1..[;]

********** Possible Correction ********** BACK or ERASE to fix.
 Insert ")" in front of [   ].
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 258

F:.......PROC;
.........DCL.I,.A(.5,1Ø).;
.........GET.LIST.(A);
.........I=.A.(.1.)

**********ERROR**********                        BACK or ERASE to fix.
 This punctuation symbol is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 258

F:.......PROC;
.........DCL.I,.A(.5,1Ø).;
.........GET.LIST.(A);
.........I=.A.(.1.)

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an arithmetic operator.
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 258

F:.......PROC;
.........DCL.I,.A(.5,1Ø).;
.........GET.LIST.(A);
.........I=.A.(.1.)

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with ",".
   Press HELP for a different suggestion.
```

```
 FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 258

F:.......PROC;
.........DCL.I, .A(.5,1Ø).;
.........GET.LIST.(A);
.........I=.A.(.[1].)

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Insert "(" in front of [   ].
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 254

G:.......PROC;
.........DCL A(1Ø),D(1Ø,1Ø),C;
.........DO  C=1 TO 1Ø;
............A(C) = D [*]

**********ERROR**********                    BACK or ERASE to fix.
 This arithmetic operator is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 254

G:.......PROC;
.........DCL A(1Ø),D(1Ø,1Ø),C;
.........DO  C=1 TO 1Ø;
............A(C) = D [*]

********** Possible Correction **********  BACK or ERASE to fix.
 Insert a subscript list in front of [   ].
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine a subscript list.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 254

G:.......PROC;
.........DCL A(1Ø),D(1Ø,1Ø),C;
.........DO  C=1 TO 1Ø;
............A(C) = D [*]

********** Possible Correction **********  BACK or ERASE to fix.
 Replace [   ] with "(".
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 254

G:.......PROC;
.........DCL A(1Ø),D(1Ø,1Ø),C;
.........DO  C=1 TO 1Ø;
............A(C) = [D] *

********** Possible Correction **********  BACK or ERASE to fix.
 Replace [   ] with a numeric operand.
   Press HELP for a different suggestion.
```

```
FILE = workspace           PL/I WORKSPACE(14-new)         SPACE = 254

G:.......PROC;
.........DCL.A(10),D(10,10),C;
.........DO..C=1.TO.10;
.............A(C).=.D.*

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [ ] with a declared variable ( numeric).
   Press HELP for a different suggestion.
```

```
FILE = workspace           PL/I WORKSPACE(14-new)         SPACE = 254

G:.......PROC;
.........DCL.A(10),D(10,10),C;
.........DO..C=1.TO.10;
.............A(C).=.D.*

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [ ] with a numeric constant.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 278

.H: . . . . . .PROC;
. . . . . . . . .DCL .J .CHAR (1) ;
. . . . . . . . .DO . .[J]

**********ERROR**********                    [BACK] or [ERASE] to fix.
 This declared variable is not permitted here.
   Press [HELP] for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 278

.H: . . . . . .PROC;
. . . . . . . . .DCL .J .CHAR (1) ;
. . . . . . . . .DO . .[J]

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with a declared variable ( numeric) .
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 278

.H: . . . . . .PROC;
. . . . . . . . .DCL .J .CHAR (1) ;
. . . . . . . . .DO . .[J]

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with a declared array ( numeric) .
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 278

.H:  . . . . .PROC;
. . . . . . . . .DCL .J .CHAR (1) ;
. . . . . . . . .DO . .[J]

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with "WHILE".
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 278

.H:......PROC;
.........DCL.J.CHAR(1);
.........DO..[J]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Insert ";" in front of [   ].
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 278

.H:......PROC;
.........DCL.J.CHAR(1);
.........[DO]..J

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with a statement.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 267

.I:......PROC;
.........DCL .A,B;
.........GET .LIST (A,B) ;
.........IF . .A+B . .[THEN]

**********ERROR**********                  (BACK) or (ERASE) to fix.
 This reserved word is not permitted here.
   Press (HELP) for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 267

.I:......PROC;
.........DCL .A,B;
.........GET .LIST (A,B) ;
.........IF . .A+B . .[THEN]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with an arithmetic operator.
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 267

.I:......PROC;
.........DCL .A,B;
.........GET .LIST (A,B) ;
.........IF . .A+B . .[THEN]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with a relational operator.
   Press (HELP) for a different suggestion.
```

```
FILE = workspace            PL/I WORKSPACE (14-new)        SPACE = 267

.I:......PROC;
.........DCL.A,B;
.........GET.LIST(A,B);
.........IF..A+B..THEN

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Insert a relational operator in front of [   ].
   Press (HELP) for a different suggestion.
```

```
FILE = workspace            PL/I WORKSPACE (14-new)        SPACE = 267

.I:......PROC;
.........DCL.A,B;
.........GET.LIST(A,B);
.........IF..A+B..THEN

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with a conditional expression.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 275

.J:......PROC;
.........DCL.I;
.........DO.I=1.TO.1Ø.[UNTIL]

**********ERROR**********                      (BACK) or (ERASE) to fix.
 This undeclared variable is not permitted here.
   Press (HELP) for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 275

.J:......PROC;
.........DCL.I;
.........DO.I=1.TO.1Ø.[UNTIL]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [    ] with an arithmetic operator.
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 275

.J:......PROC;
.........DCL.I;
.........DO.I=1.TO.1Ø.[UNTIL]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [    ] with "BY".
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 275

.J:......PROC;
.........DCL.I;
.........DO.I=1.TO.1Ø.[UNTIL]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with "WHILE".
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 275

.J:......PROC;
.........DCL.I;
.........DO.I=1.TO.1Ø.[UNTIL]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Insert ";" in front of [   ].
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 283

.K: ......PROC;
.........DCL .I;
.........FOR .[I]

**********ERROR**********·                    [BACK] or [ERASE] to fix.
 This declared variable is not permitted here.
   Press [HELP] for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 283

.K: ......PROC;
.........DCL .I;
.........FOR .[I]

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Insert ":" in front of [   ].
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 283

.K: ......PROC;
.........DCL .I;
.........[FOR] .I

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with a statement.
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)       SPACE = 283

.K: ......PROC;
.........DCL .I;
.........[FOR] .I

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with a reserved word "IF".
   Press [HELP] for a different suggestion.
```

```
FILE = workspace            PL/I WORKSPACE(14-new)          SPACE = 283
------------------------------------------------------------------------
.K:......PROC;
.........DCL.I;
.........[FOR].I

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with a reserved word "DO".
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Press (HELP) for a different suggestion.
```

```
FILE = workspace            PL/I WORKSPACE(14-new)          SPACE = 283
------------------------------------------------------------------------
.K:......PROC;
.........DCL.I;
.........[FOR].I

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Remove [   ] from the program.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 277

.L: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED;
. . . . . . . . .I .= .1 .[TO]

**********ERROR**********·                    [BACK] or [ERASE] to fix.
 This reserved word is not permitted here.
   Press [HELP] for more information.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 277

.L: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED;
. . . . . . . . .I .= .1 .[TO]

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with an arithmetic operator.
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 277

.L: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED;
. . . . . . . . .I .= .1 .[TO]

********** Possible Correction ********** [BACK] or [ERASE] to fix.
 Replace [   ] with ";".
   Press [HELP] for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 277
.L:......PROC;
.........DCL.I.FIXED;
.........I.=.1.TO

********** Possible Correction ********** BACK or ERASE to fix.
 Insert a reserved word "DO" in front of [    ].
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 281

.M: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED;
. . . . . . . . .DCL .[ ]

**********ERROR**********                         BACK or ERASE to fix.
 This entry name is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 281

.M: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED;
. . . . . . . . .DCL .[ ]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with an identifier declaration.
   Press HELP for a different suggestion.
   Press SHIFT HELP to examine an identifier declaration.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 281

.M: . . . . . .PROC;
. . . . . . . . .DCL .I .FIXED;
. . . . . . . . .DCL .[ ]

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with "(".
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 281

.M:......PROC;
.........DCL.I.FIXED;
.........DCL.M

********** Possible Correction ********** BACK or ERASE to fix.
 Replace      with an undeclared variable.
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 281

.M:......PROC;
.........DCL.I.FIXED;
.........DCL.M

********** Possible Correction ********** BACK or ERASE to fix.
 Replace      with "END".
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 279

.N: . . . . . .PROC;
. . . . . . . . . .DCL .I,A;
. . . . . . . . . .PUT .LIST .[;]

**********ERROR**********·                          (BACK) or (ERASE) to fix.
 This punctuation symbol is not permitted here.
   Press (HELP) for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 279

.N: . . . . . .PROC;
. . . . . . . . . .DCL .I,A;
. . . . . . . . . .PUT .LIST .[;]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with "(".
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 279

.N: . . . . . .PROC;
. . . . . . . . . .DCL .I,A;
. . . . . . . . . .PUT .[LIST] .;

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [   ] with "SKIP".
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 279

.N:......PROC;
.........DCL.I,A;
.........PUT.[LIST].;

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [    ] with "PAGE".
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 274

.0:.......PROC;
..........DCL.I,J;
..........PUT.LIST.(.'START.OF.PROGRAM',.[]

**********ERROR**********.                    BACK or ERASE to fix.
 This punctuation symbol is not permitted here.
    Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 274

.0:.......PROC;
..........DCL.I,J;
..........PUT.LIST.(.'START.OF.PROGRAM',.[]

********** Possible Correction **********  BACK or ERASE to fix.
 Insert a declared array in front of [   ].
    Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 274

.0:.......PROC;
..........DCL.I,J;
..........PUT.LIST.(.'START.OF.PROGRAM',.[]

********** Possible Correction **********  BACK or ERASE to fix.
 Insert an expression in front of [   ].
    Press HELP for a different suggestion.
    Press SHIFT HELP to examine an expression.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)        SPACE = 274

.0: ......PROC;
.........DCL I,J;
.........PUT LIST ( 'START OF PROGRAM' , )

********** Possible Correction ********** BACK or ERASE to fix.
 Remove      from the program.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 272

.P:......PROC;
.........DCL.A,B,I;
.........GET.LIST.(.A,.B

**********ERROR**********                  BACK or ERASE to fix.
 This entry name is not permitted here.
   Press HELP for more information.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)          SPACE = 272

.P:......PROC;
.........DCL.A,B,I;
.........GET.LIST.(.A,.B

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [    ] with a declared variable.
   Press HELP for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)        SPACE = 272
.P:......PROC;
.........DCL.A,B,I;
.........GET.LIST.(.A,.B

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [   ] with a declared array.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 267

..IT:....PROC;
.........DCL.A;
.........GET.LIST(A);
.........PUT.LIST(A);
.........STOP.[IT]

**********ERROR**********                    (BACK) or (ERASE) to fix.
 This entry name is not permitted here.
   Press (HELP) for more information.
```

```
FILE = workspace          PL/I WORKSPACE (14-new)          SPACE = 267

..IT:....PROC;
.........DCL.A;
.........GET.LIST(A);
.........PUT.LIST(A);
.........STOP.[IT]

********** Possible Correction ********** (BACK) or (ERASE) to fix.
 Replace [  ] with ";".
   Press (HELP) for a different suggestion.
```

```
FILE = workspace          PL/I WORKSPACE(14-new)       SPACE = 267

..IT:....PROC;
.........DCL.A;
.........GET.LIST(A);
.........PUT.LIST(A);
.........STOP.IT

********** Possible Correction ********** BACK or ERASE to fix.
 Replace [    ] with "END".
```

VITA

Michael Tindall was born in Seattle, Washington on September 20, 1949. He graduated from Seattle Pacific College with a B.S. in mathematics in June, 1971, and was elected to "Who's Who in America Among Students" in that year. During four years of graduate school work in computer science at the University of Illinois at Urbana-Champaign, he received the Master of Science degree (1975) as well as serving as the coordinator/ secretary of the Computer Science Graduate Student Organization and sitting on numerous departmental committees. He was elected to the Society of Scientific Researchers, SIGMA XI, in 1973, and to the National Scholastic Honorary, Phi Kappa Phi, in 1974. He is a member of the Association of Computing Machinery.

IBLIOGRAPHIC DATA HEET	1. Report No. UIUCDCS-R-75-748	2.	3. Recipient's Accession No.

Title and Subtitle

An Interactive Compile-time Diagnostic System

5. Report Date
October 1975

6.

Author(s)
Michael Harry Tindall

8. Performing Organization Rept. No. UIUCDCS-R-75-748

Performing Organization Name and Address
University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

Sponsoring Organization Name and Address
IBM Corporation
Thomas J. Watson Research Laboratory
Yorktown Heights, New York 10598

13. Type of Report & Period Covered
Doctoral Thesis

14.

Supplementary Notes

Abstracts

The following interactive diagnostic system can be devised to analyze syntax errors detected in an interactive timesharing compiling environment. The system is "automatic", that is, it is driven by the compiler's parser tables. The error system behaves like a consultant by suggesting "possible corrections" of the program to the user, and at any time the user can proceed to fix the program or request further suggestions. In addition, the "possible correction" diagnostic suggestions can refer to not only individual "tokens" in the user's program, but also higher-level constructs, such as "expressions", "array bounds", etc.

Key Words and Document Analysis. 17a. Descriptors

Compilers, error analysis, interactive compiling, parsers, transition diagrams.

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	173
	20. Security Class (This Page) UNCLASSIFIED	22. Price -------

INSTRUCTIONS FOR COMPLETING FORM NTIS-35 (10-70) (Bibliographic Data Sheet based on COSATI Guidelines to Format Standards for Scientific and Technical Reports Prepared by or for the Federal Government, PB-180 600).

1. **Report Number.** Each report shall carry a unique alphanumeric designation. Select one of the following types: (a) alphanumeric designation provided by the sponsoring agency, e.g., **FAA-RD-68-09**; or, if none has been assigned, (b) alphanumeric designation established by the performing organization e.g., **FASEB-NS-87**; or, if none has been established, (c) alphanumeric designation derived from contract or grant number, e.g., **PH-43-64-932-4**.

2. Leave blank.

3. **Recipient's Accession Number.** Reserved for use by each report recipient.

4. **Title and Subtitle.** Title should indicate clearly and briefly the subject coverage of the report, and be displayed prominently. Set subtitle, if used, in smaller type or otherwise subordinate it to main title. When a report is prepared in more than one volume, repeat the primary title, add volume number and include subtitle for the specific volume.

5. **Report Date.** Each report shall carry a date indicating at least month and year. Indicate the basis on which it was selected (e.g., date of issue, date of approval, date or preparation.

6. **Performing Organization Code.** Leave blank.

7. **Author(s).** Give name(s) in conventional order (e.g., John R. Doe, or J.Robert Doe). List author's affiliation if it differs from the performing organization.

8. **Performing Organization Report Number.** Insert if performing organization wishes to assign this number.

9. **Performing Organization Name and Address.** Give name, street, city, state, and zip code. List no more than two levels of an organizational hierarchy. Display the name of the organization exactly as it should appear in Government indexes such as USGRDR-I.

10. **Project/Task/Work Unit Number.** Use the project, task and work unit numbers under which the report was prepared.

11. **Contract/Grant Number.** Insert contract or grant number under which report was prepared.

12. **Sponsoring Agency Name and Address.** Include zip code.

13. **Type of Report and Period Covered.** Indicate interim, final, etc., and, if applicable, dates covered.

14. **Sponsoring Agency Code.** Leave blank.

15. **Supplementary Notes.** Enter information not included elsewhere but useful, such as: Prepared in cooperation with . . . Translation of . . . Presented at conference of . . . To be published in . . . Supersedes . . . Supplements . . .

16. **Abstract.** Include a brief (200 words or less) factual summary of the most significant information contained in the report. If the report contains a significant bibliography or literature survey, mention it here.

17. **Key Words and Document Analysis. (a). Descriptors.** Select from the Thesaurus of Engineering and Scientific Terms the proper authorized terms that identify the major concept of the research and are sufficiently specific and precise to be used as index entries for cataloging.
(b). Identifiers and Open-Ended Terms. Use identifiers for project names, code names, equipment designators, etc. Use open-ended terms written in descriptor form for those subjects for which no descriptor exists.
(c). COSATI Field/Group. Field and Group assignments are to be taken from the 1965 COSATI Subject Category List. Since the majority of documents are multidisciplinary in nature, the primary Field/Group assignment(s) will be the specific discipline, area of human endeavor, or type of physical object. The application(s) will be cross-referenced with secondary Field/Group assignments that will follow the primary posting(s).

18. **Distribution Statement.** Denote releasability to the public or limitation for reasons other than security for example "Release unlimited". Cite any availability to the public, with address and price.

19 & 20. **Security Classification.** Do not submit classified reports to the National Technical Information Service.

21. **Number of Pages.** Insert the total number of pages, including this one and unnumbered pages, but excluding distribution list, if any.

22. **Price.** Insert the price set by the National Technical Information Service or the Government Printing Office, if known.

SEP 10 1975

DEC 23
BOUND BY HNM

...ERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.746-751(1974
Lisp e CAI Implementation /
3 0112 088402075